A COMPILER TARGET MODEL FOR LINE ASSOCIATIVE REGISTERS

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the College of Engineering at the University of Kentucky

By
Paul Selegue Eberhart
Lexington, Kentucky

Director: Dr. Henry G. Dietz, Professor of Electrical and Computer Engineering
Lexington, Kentucky 2019

Copyright© Paul Selegue Eberhart 2019

ABSTRACT OF THESIS

A COMPILER TARGET MODEL FOR LINE ASSOCIATIVE REGISTERS

LARs (Line Associative Registers) are very wide tagged registers, used for both register-wide SWAR (SIMD Within a Register)operations and scalar operations on arbitrary fields. LARs include a large data field, type tags, source addresses, and a dirty bit, which allow them to not only replace both caches and registers in the conventional memory hierarchy, but improve on both their functions. This thesis details a LAR-based architecture, and describes the design of a compiler which can generate code for a LAR-based design. In particular, type conversion, alignment, and register allocation are discussed in detail.

KEYWORDS: computer, architecture, compiler, SWAR, cache, register, memory, alias

Author's signature: Paul Selegue Eberhart

Date: May 3, 2019

A COMPILER TARGET MODEL FOR LINE ASSOCIATIVE REGISTERS

By
Paul Selegue Eberhart

Director of Thesis: Henry G. Dietz

Director of Graduate Studies: Aaron Cramer

Date: May 3, 2019

ACKNOWLEDGMENTS

The many people who have been patient with my many distractions.

TABLE OF CONTENTS

Acknowledgments iii

Table of Contents iv

List of Figures vi

List of Tables vii

Chapter 1 Introduction 1
- 1.1 Introduction 1
- 1.2 The Memory Hierarchy 1

Chapter 2 History 5
- 2.1 CRegs 6
- 2.2 SIMD and SWAR 6
- 2.3 Tagged Architectures 8
- 2.4 Compiler-Managed Data Motion 10

Chapter 3 LARK 12
- 3.1 DLARs 12
- 3.2 ILARs 14
- 3.3 Architecture Specification 14
- 3.4 The Simulator 27

Chapter 4 The Compiler 29
- 4.1 Input Language 30
- 4.2 Register Allocation and Vectorization 30
- 4.3 The Assembler 33
- 4.4 Virtual Memory 33

Chapter 5 Results 35

Chapter 6 Conclusion 36

Appendix A LARKem 38

Appendix B lark1.aik 84

Appendix C larc 87

Bibliography 94

Vita 98

LIST OF FIGURES

1.1 The traditional computer memory hierarchy. 2

3.1 Data LAR structure 12
3.2 DLARs cache specified sections of memory 13
3.3 Instruction LAR structure 14
3.4 Data LAR Structure 16
3.5 Instruction LAR Structure 17
3.6 LARK memory instruction format 18
3.7 LARK utility instruction format 22
3.8 Behavior of `LOAD16I D3 D0 D5 0` 24
3.9 Behavior of subsequent `LOADF32 D6 D3 D0 2` 25
3.10 Performing a copy with LARs 25
3.11 Sample code compiled for MIPS and LARK 26

LIST OF TABLES

3.1 Word Size Encodings 16
3.2 Type Encodings 16
3.3 LARK Memory Instructions 19
3.4 LARK Arithmetic Instruction Behaviors 20
3.5 LARK Arithmetic Instruction Encodings 20
3.6 LARK Flow Control Instructions 22
3.7 LARK Utility Instruction 22
3.8 DLAR1 Contents 23
3.9 Instruction Count Comparison for Sample Code 26

Chapter 1 Introduction

"Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."
-John Backus
ACM Turing Award Speech, 1977

John Backus was focused on the effects of the Von Neumann design on the programmer [1] - however, the matter is fundamentally a problem of hardware design. This document proposes an architecture based on Line Associative Registers, discusses the implications of that design, and situates it in terms of other efforts to ameliorate the bottleneck.

1.1 Introduction

LARs (**L**ine **A**ssociative **R**egister**s**) are a memory structure designed to be used as the upper levels of the memory hierarchy in a new class of architectures. The chief objective of LARs-based designs is to provide a general-purpose model of computation in which memory accesses are minimized by explicitly managing a large pool of data and meta-data at the top of the memory hierarchy. Line Associative Registers fill the role of both registers and caches in a traditional memory hierarchy, bringing many of the advantages of each while avoiding their more egregious faults. This work chiefly deals with the development of software tools for programming a LAR-based design, but also details new advancements in the architecture, and software simulation tools for testing these tools and features.

1.2 The Memory Hierarchy

In the majority of recent computer designs, memory is laid out in an increasingly deep hierarchy, with a small, fast memory near the CPU at the top in the form of registers, a series of increasingly large and slow caches, and eventually a large DRAM bank for main memory. There is also typically an option to page inactive segments of the DRAM bank to slow secondary storage.

At the top of the memory hierarchy is a small register file consisting of no more than a few kilobytes of extremely fast memory, typically SRAM. For example, a naive implementation of the x86-64 architecture common in modern PCs would feature six-

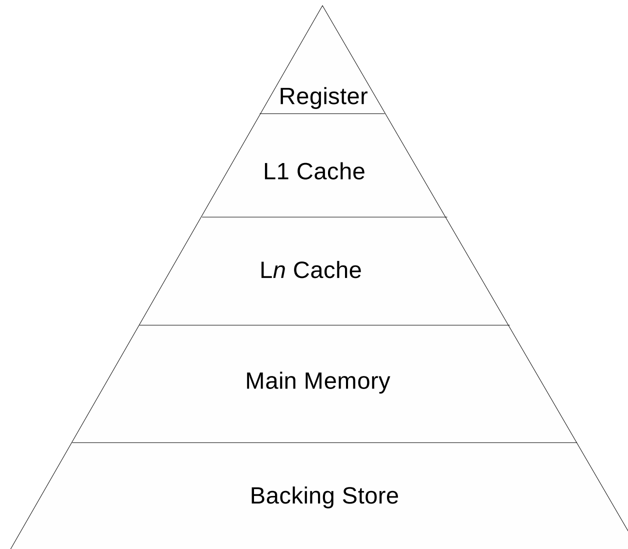

Figure 1.1: The traditional computer memory hierarchy.

teen 64 bit integer registers, and an additional sixteen 128 bit floating point registers, for a total of only 384 bytes of addressable register file.

Even the visible register files have grown considerably in size. For example, Intel's traditionally register-starved x86 architecture has sprung a series of extensions: a series of width extensions bringing the sixteen basic registers from sixteen to thirty-two, and subsequently 64 bits, eight 80-bit x87 floating point registers, and eight 64-bit vector registers added with the MMX instruction set, and extended at lest three times, once with SSE, to 128 bits, again with the transition to x86-64, to 16 registers, and again with AVX [2], to 256 bits per register.

At the very large end for designs in common use, Intel's Itanium architecture [3] features 128 64-bit registers, 128 82-bit floating point registers, and an additional 64 one-bit predicate registers, for a total of 2344 bytes of addressable register file. This propensity for extremely small register files has a number of valid rationale: high speed memory, typically SRAM on the same die as the processor functional units, is extremely expensive in terms of die area and hence cost, and the simple reality that it is much easier to sustain fast addressing on smaller memories.

The register file in almost all computers is addressed separately from main memory, which creates problems with *aliasing*; the inability to determine whether indexed accesses point to the same location, creating unnecessary traffic over the memory bus to guarantee synchronization after changes. In modern systems, the register file will often support *renaming*, which allows for a larger number of physical registers than are specified in the instruction set. This allows the system to keep more data resident at the top of the memory hierarchy, as well as opportunities to hide or eliminate false dependencies, while maintaining instruction-level compatibility. This practice comes at a cost of significant complexity and power consumption, and is opaque to the software.

Many designs considerably extend the size of their register file internally with various register renaming schemes, however, this extended space is available only for automated micro-optimization, rather than programmer or compiler managed optimizations, as it is not explicitly manageable and varies wildly between implementations of the same ISA.

Typical modern computers use a large bank of DRAM as main memory. This DRAM bank is larger, slower, lower power, and far less expensive than SRAM used for the upper levels of the hierarchy, and in many early computers. A large DRAM is necessarily slower than smaller static memories because it requires a relatively complicated decoding scheme to address and route from a large memory, as well as timing instability due to required refresh cycles. Furthermore, in a system supporting virtual memory (paging or segmentation), there will be a translation layer (page table) mapping between virtual and physical addresses which adds further delays and timing variability to accessing main memory. To improve performance without dramatically increasing costs, a system of smaller, faster caches is placed between the main memory and the processor. These caches are generally on the order of hundreds of kilobytes to a few megabytes, and may be managed by a wide variety of increasingly sophisticated automatic mechanisms, and are most often implemented in SRAM. Because the cache represents an extremely small fraction of the system memory, any automated replacement scheme is susceptible to unpredictable and/or pathological conditions, in which the access pattern and the replacement scheme interact unfavorably. Finally, caches exacerbate the nondeterministic timing of memory accesses; a datum fetched from cache will arrive orders of magnitude faster than a datum fetched from main memory, and it is impossible (or, at very least, impractical) to accurately statically predict whether a particular datum will be in cache prior to a particular read. Analogous to the use of caches to attempt to improve memory fetch performance, the address translation process is typically also cached, employing a translation lookaside buffer (TLB) to cache probable page table lines in a smaller, faster memory close to the CPU. Also like the caching of data, the TLB accelerates properly-predicted address translations (hits), but further slows down misses, and introduces a large variability in the time it will take for a memory access to complete.

The Register-Cache-Memory hierarchy is not the only memory layout which has seen widespread use. A variety of computer designs employ a small *scratchpad memory* [4]. Scratchpads are small, fast memories, generally of the same scale and technology as Caches, but are explicitly managed. This allows scratchpads to be managed

with far more accurate but computationally expensive static techniques, which can, in theory, dramatically improve their utility relative to a cache of the same size. Unfortunately, because a scratchpad is explicitly managed, it requires that the scratchpads must be consistent in size and access behavior among compatible machines, or code must be recompiled to specifically target the different sizes of scratchpad.

There is a common addendum to Moore's law [5] noting that while the speed of central processing units tends to increase at an exponential rate, DRAM speeds and sizes have only grown roughly linearly. This has created a large disparity of perhaps 400 clock cycles between the speed of a system's CPU and main memory, which has been the impetus for the ever deeper memory hierarchy.

Copyright© Paul Selegue Eberhart, 2019.

Chapter 2 History

The LARs design has its roots in a variety of previous projects. It descends most directly from three existing areas of research: CRegs [6], a type of tagged registers, SWAR (**S**IMD **W**ithin **A** **R**egister) designs [7] such as Intel's SSE, and compiler-managed memory hierarchies [8].

Research into LARs began with the master's thesis of Krishna Melarkode [9], in 2004, and has since developed in scope and complexity, to include a number of hardware verification models, designs for software tooolchains and simulators, and a considerable amount of theoretical work into the implications of such a design.

LARs-like features have appeared in many recent commercial designs - wider, more comprehensive SIMD extensions like Intel's AVX [2] approach a general-purpose SWAR architecture. Likewise, a weak implementation of an explicitly managed tagged file can be found in the Itanium architecture's ALAT (Advanced Load Address Table)[3]. However, in insuring that the design failed safe if improperly managed, many of it's potential benefits have been lost.

Similarly, the idea of decoupling of instruction fetch and execution to confront the Von Neumann bottleneck has been in practice since at least the mid 1960s. Most of these efforts, such as the use of scoreboarding in the CDC6600 (1964) and Tomosulo Algorithm in the IBM S360 Model 91 (1967) have been primarily focused on the efficient use of execution units, and only tackled memory access behavior insomuch as they have been required to to meet their goals. More explicit, generalized decoupling of fetch and execution has also appeared in a number of designs, perhaps most radically in the CSPI MAP 200 array processors of the early 1980s[10]. These designs, however, tend to, in one designer's own words "place a great deal of responsibility for resource scheduling and interlocking on software." [10], and practicable implementations are largely a question of re-automating these issues into automatic (typically hardware-driven) mechanisms.

A major issue in working with LARs, or other architectural changes that significantly alter long-held assumptions, is that they are not readily miscible with current practice. An incremental approach to introducing LAR-like features is impossible as effectively utilizing LARs requires fundamental changes to accepted practice. Efforts to introduce them to existing designs or apply existing tooling is doomed to be awkward and ineffective; the extent of this incompatibility was not fully appreciated at the beginning of this project. In fact, as discussed below, one attempt to employ a structure very similar to a LAR in the form of the Itanium ALAT has already failed, likely because it was compromised to improve compatibility with a conventional memory hierarchy. It is hoped that this research will yield concepts and techniques which which will be necessary to build computer systems employing LARs or similar structures.

2.1 CRegs

A direct predecessor project to LARs was cache-registers, CRegs [6]. CReg is pronounced "C-Reg," and is a portmanteau of "Cache Registers." CRegs are scalar storage elements that operate in their own distinct address space, much like conventional registers. The primary CReg contribution is the load-time addition of address tags to datum near the top of the memory hierarchy. CRegs were introduced in 1989 by Henry Dietz and Chi-Hung Chi, with the primary intention of providing hardware support to resolve *ambiguous aliasing*. Ambiguous aliasing is the situation in which it is impossible to statically determine whether two names refer to the same value. Ambiguous aliases present a significant problem for systems that rely on conventional registers, as every aliased datum must be flushed from the register to a level of the memory hierarchy where the source addresses can be compared any time any aliased value is written to, creating large, correlated, and unpredictable memory traffic.

To borrow an example from the original CRegs publication, the snippet of pascal-like pseudocode below is a trivial case of an ambiguous alias.

```
readln(i,j);
b := a[i]+a[j];
```

Because `i` and `j` are read from user input, it is entirely impossible to statically determine if `a[i]` and `a[j]` refer to the same memory location, and so they must either be operated on in-memory, or, if `a[i]` and `a[j]` have been loaded into registers, both be evicted from their registers and re-read whenever either is written to in order to maintain consistency. CRegs solve the problem of ambiguous aliases in all cases, by use of an address tag on each element, and associatvity in the storage elements. LARs retain the address tag and associativity properties of CRegs, but extend them with vector behavior from another contemporary memory technology, described in the next section.

2.2 SIMD and SWAR

Another major predecessor technology for LARs is SWAR (**S**IMD **W**ithin **A** **R**egister). SWAR is a form of low-level parallelism, in which SIMD (**S**ingle **I**instruction **M**ultiple **D**ata) operations are performed on collections of data packed into a single wider register. Like the vector machines they are derived from, SWAR instructions enable machines to perform operations over a set of similar data. As these extensions develop, they often lead to the introduction of wider registers into to the datapath of the enhanced processors. Most SWAR implementations are extensions to an existing scalar architecture, such as the AltiVec extensions to PowerPC, or NEON extensions to ARM. Intel's x86 platform has endured an entire series of SWAR additions to the instruction set, starting with MMX in 1997, and accreting no fewer than 17 sets of additional vector instructions since.

This after-the-fact design model comes with a variety of implications and compromises in the functionality and generality of SWAR extensions.

Most implementations of SWAR extensions have extensive, and sometimes curious, limitations. For example, Intel's MMX contains instructions for adding eight 8-bit, four 16-bit or two 32-bit signed or unsigned integers packed into a single 64-bit register, and an instruction for multiplying four 16-bit integers packed into a 64-bit register, but not eight 8-bit integers in the same configuration [11]. Likewise, there are generally strict alignment constraints on SIMD extensions, both for in-memory structures which interact with the cache hierarchy to speed or slow loading the vector registers, and on the contents of the vector registers themselves [12], which tend to create additional bookeeping operations and memory traffic in order to use the vector instructions. Worse, the practice of using different register sets for scalar and vector operations exacerbates the aliasing problems discussed above, forcing loads and stores simply to maintain consistency when data is proximally handled by both the scalar and vector hardware.

Even systems designed primarily around custom very-wide SIMD engines have tended to rely heavily on conventional (and typically commodity) scalar processors to run the host operating system. These heterogeneous designs offer several advantages, especially in reducing the required complexity and keeping scheduler noise from the operating system away from the high-performance vector processors. These system architectures also force several design decisions, such as requiring special language and platform support to program and schedule the vector processors, which results in certain limitations, such as an awkward split memory space, and limiting the applicability of code developed against them to comparatively rare systems hosting the same variety of vector coprocessor.

For example, perhaps the archetypal accelerator-based machine, the Ardent Titan [13], relied on vector processors whose vector register file held 8192 64-bit values (addressable as anything from one vector of length 8192 to 32 vectors of length 256) in 1988. It, however, used a MIPS R2000 (and later R3000) processor per vector unit to run the operating system, and used the vector units only for compute offload. Being marketed as a graphics workstation, this design rather directly predicts the eventual evolution of systems with conventional host processors and attached GPUs.

Despite vector extensions being a somewhat awkward bolt-on, modern performance tuning and optimization guides suggest using the vector extensions, often almost to the exclusion of the host scalar instruction set [12]. Similarly, recent trends in high performance computing have concentrated on various derangements of vector co-processor, such as GPUs, whose pipelines are very SWAR-like.

LARs are designed with a modified SWAR model as the *primary* mode of computation, with designed-in accommodations for scalar operations. This corresponds to the ideal/desired behavior for many computing applications, where some amount of scalar operations for addressing and edge cases must be interspersed with the vectorizable primary computational work to be performed in a SIMD execution mode. Unlike SIMD coprocessor designs, LARs allows these interspersed instructions to be performed on the same, without any expensive memory transfers between host and coprocessor, realigments or corner-turns to suit the relative alignment constraints of the host and coprocessor or extension, or aliasing/timing concerns between data touched by the scalar and vector execution units. Likewise, LARs accept arbitrary

base addresses, reducing the amount of effort required to massage data into position for vector operations.

Packed SIMD/Vector Registers

Perhaps the closest current designs to LARs are efforts to add flexible vector extensions to modern ISAs. The dominant example, RISC-V's proposed Vector Extensions [14] offer a sort of generalized length-independent vector support instead of fixed-width SWAR. These designs make a number of decisions which place them in a slightly different design space than traditional vector machines, SWAR extensions, or LARs. The RISC V proposal does employ something resembling tagged registers; a set of configuration registers specify the type and vector length of items stored in a re-configurable register file, and must be appropriately manipulated before using the vector units. Like LARs, this allows for polymorphic instructions based on the tags. A major distinction in this design is that the lengths (in operands) of vector operations are also determined by a vector length register, set with `setvl`, essentially providing hardware-parameterized vectors suitable for allocation via strip-mining [15]. Unfortunately, these vectors do not have any form of alias analysis or scheduling support beyond simple predicates, so they can not tolerate dependencies between data in a vector operation. Worse, because of the parameterization, inherently scalar, or at best mapped onto a multi-issue pipeline, in actual execution.

2.3 Tagged Architectures

A final major influence on LARs design are tagged architectures. Tagged architectures are computer architectures in which metadata, such as type information, is stored with the data, rather than inferred form the instructions used to manipulate that data [16]. Tagged architectures offer a number of theoretical advantages over conventional designs, and historically incur a number of practical drawbacks, which LARs attempt to eliminate.

To enumerate a few of tagged architectures' purported advantages, tags provide an opportunity for a smaller, simpler instruction sets, as they enable generic instructions. This allows for more compact instruction encodings and, in principle, simpler code generation and greater code reuse. Tags also enable unusual addressing modes, such as field-and-offset to allow word-oriented machines to address bytes [17]. Tagging also provides automated type checking and conversion, making most operations inherently type safe, and avoiding complicated subroutines to convert between types. Many tagging systems also enable data protection, by marking data with some form of access descriptor.

Tagged architectures have generally fallen into two categories; fixed tagging, in which each memory location is associated with a tag, and distributed tagging, in which each logical object is associated with a tag. Fixed tagging allows tags to be retrieved in a regular pattern, that is without additional address calculations, while dynamic tagging allows for fewer tags and tagging of irregularly sized objects. However, both systems still require that tags be moved in and out of main memory along with the

objects they describe, creating additional memory traffic through the Von Neumann Bottleneck, which is increasingly untenable as increases in processing speed continue to outstrip improvements in memory access.

A number of commercial designs have employed tagged architectures, most famously the Burroughs Large Systems family [18](1961), IBM System/38[17] (1979) and SWARD [19], LISP machines [20] (1981), and Intel iAPX 432 [21] (1981). While several of these machines were commercially successful, they were all eventually outcompeted [16] by simpler untagged designs. Even those machines developed expressly to execute dynamically typed languages, like the LISP machines, which have the most to gain as they must constantly perform type checking at runtime, were eventually superseded by simpler untagged designs and software support. This decline has given tagged architectures something of a bad name in modern times, particularly by association with Intel's iAPX project [21], which was an high-profile expensive failure [22], due in part to its use of a tagging system. The iAPX tags were exceptionally long - it employed 128-bit object descriptors and 32-bit access descriptors which had to be handled during almost all memory accesses [17].

The Burroughs Large Systems offer some similarities to the tags proposed for LARs. The B5000 [18] reserved a single bit of each (48 bit) machine word, to distinguish "control" and "numeric" data, though it did not use this tag system in the case of character data or code. The B6500 and later expanded to three bits of tag, which allowed for a considerably richer description; for example, the low bit is used to distinguish words which belong to the system state (low bit set) from those which belong to the user code (low bit unset). While the Burroughs Large System architecture was a stack machine, and their tag system was primarily as a security measure, comparable to a more fine-grained NX (No-Execute) bit found in the paging hardware in many modern designs, these latter members of the family also used their tags to distinguish data types. Another historical machine whose memory handling can be used as a point of reference for LARs is the Harris H500/800 family [23] and their progenitor the Datacraft 6024, marketed in the late 1970s. The Harris' distinguishing feature was its sophisticated-for-the-time cache-backed virtual memory system operating on 48-bit words. Like the IBM System/38 above, these machines were word-oriented, and used their tagging system to allow for subword addressing when operating on byte or 24-bit word oriented data. LARK is designed with a generalization of this idea, in which memory accesses are always line-at-a-time, and the low bits of addresses for smaller data objects are performed by treating the low bits of addresses as offsets into a LAR.

The major weakness of historical tagged designs is that the addition of tags in main memory creates additional memory traffic with each datum. As computer performance in the modern era is typically constrained chiefly by the memory system, this is an unacceptable compromise. Another major problem for tagged architectures is language support; most general-purpose programming languages do not offer native facilities that match up with the tags on a particular architecture, which results in stilted use of tags and/or additional load on the toolchain to determine the mapping. Relatively successful tagged architectures have tended to be co-developed with a programming language; n the case of the LISP Machines, the machine was designed

specifically to support the needs of the existing LISP language family. Similarly, the System/38 uses a layered ISA, where the user-facing software toolchain produces a high-level intermediate language, which is then translated by what we would now refer to as a firmware layer [17] - in many ways presaging the now-common design of dynamically translating a high-level ISA into microps in microcode.

The tagging employed in LARK, and intended for use in any LARs based architecture, is different than any of the prior designs - instead of tagging in memory, LARK's `LOAD` and `STORE` instructions set and manipulate tags on data in the DLAR file, giving the benefits of a tagged architecture, without incurring the penalty in memory traffic suffered by designs which maintain tags in RAM. To do this, data is tagged when loaded into a LAR from main memory, and the tags remain attached to the data in the LAR file even if moved or duplicated, but are never written back to main memory.

2.4 Compiler-Managed Data Motion

There are a number of approaches present in commercial designs to allow the software toolchain to guide or control the movement of data up and/or down the memory hierarchy, rather than relying directly on automated, heuristic cache management.

The general solution to compiler-managed control of fast memories is to simply expose the fast memories to the architecture as a memory segment - typically the small fast memories in this arragement are called "scratchpads." The most recent successful architecture to use a scratchpad was the SPU in the IBM Cell processor [24], though it is certainly not unique in this arrangement. This explicitly-managed separate memory offers a number of challenges. The scratchpad memory itself must be managed with additional instructions. The size of the scratchpad is exposed in the ISA, which restricts portability; a larger scratchpad cannot be utilized without altering the code, as is possible with caches. One feature which is neither a clear advantage or disadvantage is that while in a cache hierarchy, data in the cache is a copy of data in main memory; while in a scratchpad, the data in the scratchpad is not backed by main memory unless expressly transferred.

The most common solution to offer limited software control of the memory hierarchy are the cache control instructions are present in the majority of modern architectures. A typical example is the x86 family `PREFETCHh` and `CLFLUSH` [25] instructions, which hint a promotion of the line containing a data byte up the cache hierarchy, and invalidate a line evicting it from the entire cache hierarchy respectively. These instructions do not offer complete programmatic control of caches; the cache control logic can 'choose' to ignore a promotion hint, and cache occupants must still obey the associativity rules of the cache, so promotions may inadvertently evict needed data.

Prefetching is also something of a leak in abstraction layers, exposing implementation-specific details and behaviors which "do[es] not affect program behavior" [25] at the level of the ISA. This variation among implementations also makes the effective use of prefetch instructions non-portable in much the same way as explicitly managed memory structures, like scratchpads, leading to the same concerns about requiring recompilation for specific targets.

There is some variety in the cache control instructions offered among ISAs, PowerPC in particular has a relatively powerful set of cache control primitives exposed through a mixture of instructions and status registers. The PowerPC cache controls allow a range of interesting features, and in fact the cache hierarchy is typically in a disabled state until initialized by software [26]. Configuration includes completely disabling particular instruction or data caches to avoid coherency contention, manipulating manipulating the prediction policies among a set of platform-defined alternatives, whole-cache or specific-way locking, and separate flushing and invalidation, the use of which incurs state-tracking within the program, as there is hardware-enforced protection to prevent the flushing of data which has been modified but not written back, a complication not incurred by LARs.

Like most designs, typical PowerPC cache hierarchies have specific instruction and data caches for L1, but shared caches for higher levels. Unlike most designs, PowerPC cache control allows for control of how the shared caches are used, such as marking the entire L2 cache as an instruction cache, and forcing all data traffic onto the bus.

Speaking in generalities about ARM caches is not productive because there is an extensive diversity of cache hierarchies employed in ARM compatible designs [27].

Prefetching can also be performed automatically. A downside of automatic prefetching in most modern architectures is the risk of poisoning caches with speculative loads; because most cache mechanisms are automated and associative, an incorrect speculative load may not only generate spurious memory traffic, but evict a piece of data which was actually needed from cache. This may cascade into more than a single fetch of stall, for example if the evicted data was both dirty and immediately needed, the process will have to wait as the line is written back, then read again. Worse, if the data in the page table was used for indirection, whole chains of otherwise unnecessary memory accesses may be triggered as accesses cascade. LARs avoid this entirely by exclusively explicitly managing the memory hierarchy, such that "accidental" algorithmic eviction is not a threat.

The structure most similar to LARs in a modern commercial designs is the **A**dvanced **L**oad **A**ddress **T**able (ALAT) present in the Intel Itanium architecture [28]. The ALAT is a 32-entry associative memory which is used to perform speculative data loads. An `ld.a` instruction can be generated by the compiler, which will produce an entry in the ALAT listing the source address, size, destination register, and state of the load. In the Itanium architecture, the ALAT is used as a supplement to a conventional automated cache memory hierarchy, which allows a limited degree of compiler-guided pre-fetch without risking poisoning associative caches with speculative loads. Results obtained by modifying a compiler to perform as much speculative loading as possible indicate that compiler-managed (pre)fetch, even driven by naive but extremely aggressive speculative loading, will perform well under most circumstances, and cause only very rare, minor performance regressions [28].

LARK, like any realizable LAR-based architecture, relies *entirely* on statically scheduled prefetching for all instruction loads - and all data loads.

Copyright© Paul Selegue Eberhart, 2019.

Chapter 3 LARK

This chapter attempts to explain both the general properties of LAR-based designs, and specifically discusses the LARK architecture, a straw-man design used across the current generation of LARs research. The first two major sections are dedicated to explaining the LAR structures themselves - DLARS (**D**ata **L**ine **A**ssociative **R**egisters), and their simpler peer, ILARS (**I**nstruction **L**ine **A**ssociative **R**egisters)

3.1 DLARs

The defining element of a LARs-based design is the DLAR. A file of DLARs fills the roles of both the register file and data cache in a fully LAR-based architecture. Each DLAR consists of a wide data field, on the order of kilobits, and a set of meta-data fields, which include a source address, dirty bit and type information to indicate the interpretation of the data field. Type tagging at the register level is a best case compromise. It allows for an extremely regular instruction set via context sensitivity, and eliminates the need for explicit type conversion instructions. It also avoids the common problems of type tagged architectures, such as those famously experienced by Intel's iAPX432 [21] which employed a fully tagged object oriented memory system, and in turn created a massive penalty in terms of memory traffic, exactly the thing LARs are intended to minimize. A generic diagram of a file of DLARs is shown below.

LAR NR	Data	Address		WDSZ	TYP	D
		TAG	OFFSET			
	2^m blocks	$n-m$ bits	m bits			1 bit
D0						
D1						
D2						
...	...	...	...	...	...	...
Dxx						

Figure 3.1: Data LAR structure

A LAR file of useful size requires a considerable quantity of fast memory to implement. However, LARs can replace both caches and registers in the memory hierarchy, and with features such as register renaming and the continuing growth of caches, require no more high-speed memory than a conventional modern design.

The tagging in LARs is, again, dissimilar to most predecessors in that it performs tagging at load time, and tags only the register file, not the main memory, thus avoiding the space inefficiency and, more importantly, additional memory bus bandwidth of a tagged-in-memory system. The address tags on DLARs importantly allow for associativity; two DLARS pointed at the same base address will present a consistent view of their contained data.

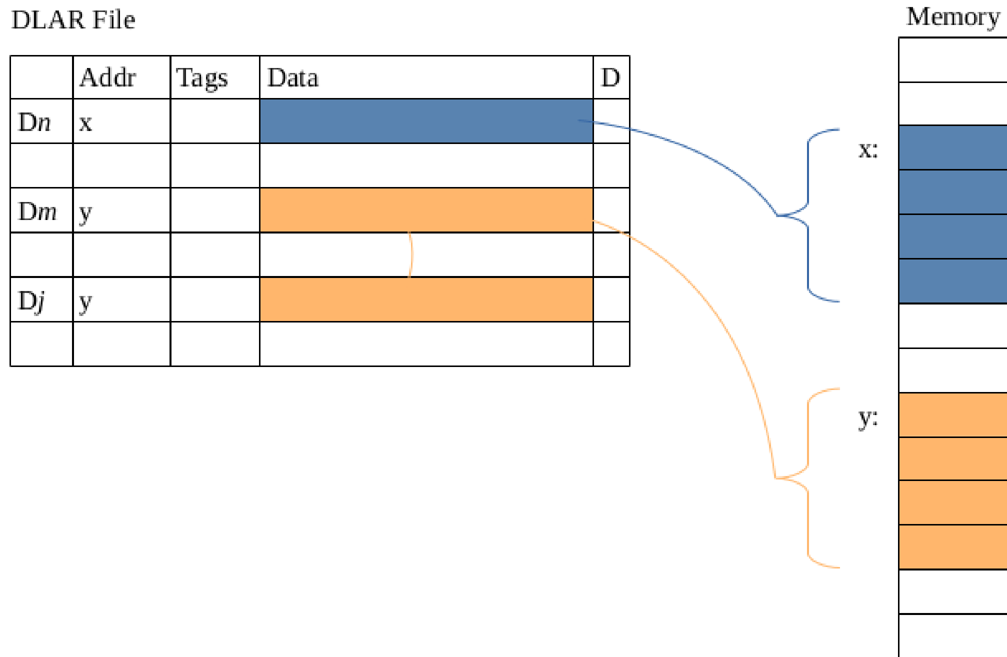

Figure 3.2: DLARs cache specified sections of memory

In use, DLARs are best thought of as annotated windows into main memory. A DLAR, once loaded, caches a line-size segment of main memory, to which it guarantees extremely fast access (reads and writes) through a convenient handle (the DLAR number). It also provides type annotations to simplify specifying operations inside the window. The associativity property of DLARs guarantee that any change made to the DLAR-masked area of memory will be consistent; all accesses to that range will necessarily go through the DLAR file, so there is no risk of aliasing. The version of the data in main memory is *eventually* written back via the lazy write-back mechanism, but particular changes which are subsequently overwritten while still resident in the DLAR file may never reach main memory. An illustration of this behavior is illustrated in figure 3.2, showing one DLAR caching the memory block at base address x, and two caching the same block at location y.

It would be possible but likely unwise to combine a LAR file with caches; it is in principle possible for a designer to back a LAR file with a cache. However, caches are most likely not a good use of circuitry in a design with a LAR file - the relatively low effective utilization and added data life-cycle complexity do not make caches a practical use of chip area in a design already equipped with LARs. Furthermore, adding a cache to a LAR-based system would likely ruin the ability to make reasonable static scheduling decisions - statically scheduling instructions around memory timing is already difficult, effectively statically scheduling instructions in the face of order-of-magnitude nondeterminism in memory access timing is impossible, a

problem experienced by existing VLIW designs.

3.2 ILARs

The other possible major use for a LAR-like structure is in the instruction path. The requirements for a LAR in the instruction path are slightly different than those in a data path, and thus practical implementations of LAR based architectures will require differently-structured LARs for use in the instruction path. The basic LAR structure described earlier is appropriate for a Data LAR (DLAR), while ILARs (Instruction LARs) hold only blocks of instruction-sized units, are indexed by the program counter, and are not directly mutatable, removing the need for most of the metadata fields.

LAR NR	Data	Address
	linesize	*addrsize*
I0		
I1		
I2		
...	...	...
D255		

Figure 3.3: Instruction LAR structure

An useful side-effect of the line oriented nature of a LAR-based memory hierarchy is that in-memory instructions could easily be stored in a compressed form. ILAR-sized could reasonably be compressed in-memory, and decompressed at load time, saving both storage space and memory bandwidth. This adds a minor complication in that it violates the assumption of an injective mapping between the contents of the LAR file and the contents of memory, and a major complication in that constant-time decompression, ideally with a constant ratio, would be required for this to be practicable. Exploring and designing the compression scheme and load mechanism is roughly an MS worth of work in itself - a thesis exploring exactly that already exists in the work of Nien Lim [29]. Others have explored this sort of block-oriented instruction compression in the context of VLIW architectures [30] with moderate promise. Compressing ILARs also creates restrictions on self-modifying or otherwise dynamically generated code, though not a significantly higher barrier than Harvard architectures' separate instruction and data memories, or even the separate instruction and data caches found in most modern designs.

3.3 Architecture Specification

A major porition of this work is the development of a straw-man design, called LARK (**L**ine **A**ssociative **R**egister architecture from **K**entucky, also a pun on "on a lark") which is a large and complete enough architecture to provide a practical common

target for research and development, but be no more complicated than is necessary for that function.

The LARK architecture is specified to allow software tools, such as simulators, compilers, and system support code, to be written for a LARs-based design. LARK is intended to be small and simple to use, and is specifically designed to privilege easy leveraging of LAR advantages over supporting either common practices or hardware implementation concerns. LARK features 64-bit direct memory addressing, a dozen native data type primitives, and only 18 instructions in three basic formats. The instruction set is somewhat richer than it first seems, as may of the basic instructions can be modified with type flags and even more unusually, a vector/scalar switch bit.

LARK does not support a number of features expected in a modern architecture, such as privileged execution, virtual memory, or specific input/output management. While most of these features could be straightforwardly amended into the design, input/output will require additional special consideration, as the native lazy write-back semantics would need to be selectively defeated, though that is not particularly complicated by the standards of modern [IO]MMU behavior. Nor does LARK include some of the possible advanced features of a LAR-based design, eschewing enhancements like instruction compression in favor of a simpler, more general design.

Memories

The most distinctive feature of LARK is its storage elements; a pair of LAR files which supplant both registers and caches as the sole architectural memory element. LARK contains two 255 entry LAR files, on for instruction and one for data, which with their metadata contain 1082624 bits (roughly 132 kilobytes) of memory, which is at once quite large as compared to named architectural register files, and extremely modest when compared to renamed register files or caches. Hierarchically below the LAR files, there is a conventional 64-bit address space, without virtual memory, as exposed by a commodity memory controller and RAM modules.

DLAR File

The primary distinguishing structure of LARK is its data LAR (DLAR). DLARs contain a 2048-bit data field, a 64-bit address field, two bits each to specify the size and type of each datum in the data field, and a dirty bit. LARK has 256 such DLARs numbered D0-D255, comprising $256 * (2048 + 64 + 2 + 2 + 1) = 541952$ bits, roughly 66 kilobytes of memory for the DLAR file.

The type and word-size fields hold the values in table 3.1, which are always encoded in the same way where they appear elsewhere in the ISA, such as in instruction encodings.

The nonstandard IEE754-style 8 bit quarter-precision float is encoded as a $(1 + 4 + 3)$ - one bit for the sign, four bits for the exponent, and three bits for the mantissa. It is otherwise extended to match IEE754-style encoding; a bias of $(2^4)/2) = 7$ is applied to the exponent, an all-0 exponent is understood to denote a zero value,

LAR NR	Data	Address		WDSZ	TYP	D
		TAG	OFFSET			
	2048 bits	$n-m$ bits	m bits	2 bits	2 bits	1 bit
D0						
D1						
D2						
...	...	...	...	...	...	...
D255						

Figure 3.4: Data LAR Structure

Table 3.1: Word Size Encodings

Value	Object Size
00	8
01	16
10	32
11	64

Table 3.2: Type Encodings

Value	Type
00	Reserved
01	Unsigned Integer
10	Signed Integer (2's compliment)
11	Float (IEEE754-ish)

and an all-1 exponent is ∞ with a zero mantissa or NaN with a non-zero mantissa. This atypical format is included partly for the sake of orthogonality, and are likely of limited utility, but also act as acknowledgement of the variety of algorithms appearing in fields like machine learning, in which very fast, high-range, low-precision calculations are desirable.

The Address field is a single value, but is dynamically reinterpreted into a base and offset based on the value of `wdsz`; when `wdsz` is set to 8 (00) the low 8 bits are treated as a word offset to index each of the 256 words in the line, likewise, when `wdsz` is set to 64 (11), only the low 5 bits are understood as offset to index each of the 32 words in the line.

ILAR File

The second architectural memory of LARK is its Instruction LAR (ILAR) file. The ILAR file is also composed of 256 2048-bit data fields supplemented with metadata, making up 540672 bits (again, roughly 66 kilobytes) of memory. However, ILARs are structurally simpler and include only the data and address fields; the contents of an ILAR is always assumed to be 32, 64-bit instructions, so the type and word-size fields are unnecessary. Likewise, ILARs are not directly writable, so the dirty bit can be omitted; the non-writability of the active instruction memory (the ILAR file) is one of several non-Von Neumann properties in a strictly LAR based architecture.

LAR NR	Data	Address
	2048 bits	64 bits
I0		
I1		
I2		
...	...	...
D255		

Figure 3.5: Instruction LAR Structure

Instruction Set

The LARK instruction set consists of 54 instructions broken into four groups: Memory, Arithmetic, Flow Control, and Utility. Opcodes are eight bits, and encodings are chosen for simplicity, readability, and orthogonality rather than compactness. This is in keeping with LARK's design as a proof of concept architecture - simple implementation and easy manipulation by humans are valued over compactness or performance.

Memory

The LARK instruction set contains only two kinds of operation loads and, somewhat misleadingly, stores. These basic instructions are modified by a byte of type data used to set the `type` and `wdsz` fields of the destination LAR. The `load` instructions, intuitively, load a 2048-bit block of data from memory into a DLAR, and set the `address`, `type`, `wdsz` fields of the target LAR. DLARS are always loaded aligned to a DLAR-width, but the low bits of the address can point anywhere inside the line as an offset. The `store` instructions, less intuitively, do not trigger a writeback to main memory, but *change the tags* on a target DLAR. The only writeback mechanism in LARK is lazy; the DLAR file is continuously scanned, and the next DLAR with the dirty bit set is written back during idle memory bus cycles, giving a simple round-robin opportunistic write-back scheme. There is some possibility that certain code sequences with rapid load-and-write to widely spaced addresses could eventually force a stall when DLARs targeted for loads are marked dirty, but this scheme is sufficient

for evaluation. Different LARs experiments have handled attempting to overwrite a dirty DLAR differently. The simplest scheme, as specified for LARK, is to stall, leaving guaranteed free memory bus cycles until the round-robin writeback system clears it. This has the potential advantage of reducing pressure on the memory bus when it has become congested, but the disadvantage of potentially long stalls of unknown length as the round robin write back comes around to the desired DLAR. Another option is to "cut" the dirty DLAR being written to the front of the write back order, bounding the stall to the time for a single extra memory write, but risking more frequent stalls. More sophisticated options, such as buffers or queues for reads and/or writes offer advantages for scheduling, and this was the method employed in the LOON LARs demonstrator [31]. To determine the most advantageous scheme for attempting to load to a dirty DLAR would require analyzing , which is not yet possible.

Memory instructions are prefixed with `0b01`, followed by a `0b00` for loads and a `0b01` for stores. The next two bits encode the type of the tag to be set on the target DLAR, with the same patterns used in the `TYPE` field of a DLAR as documented in 3.2. The final two bits encode the wordsize for the contents of the target DLAR, analogously using the same encodings as the `wdsz` field of a DLAR as documented in 3.1. This scheme not only uses the same encodings for instruction fields as for corresponding DLAR fields, but creates a conveniently compact and human readable sequential numbering within the groups of instructions.

OP	DST	SRC1	SRC2	IMM
8	8	8	8	32

Figure 3.6: LARK memory instruction format

`OP` - Opcode Field - 8 Bits.
`DST` - Destination LAR - 8 Bits.
`SRC1` - First Operand Source LAR (Address field of this LAR used as base address for load) - 8 Bits.
`SRC2` - Second operand source LAR (Data field of this LAR used for effective address calculation) - 8 Bits.
`IMM` - Immediate value, added for effective address calculation (Signed) - 32 Bits.

The addressing mode is as follows:

```
Base Addr = SRC1.Address + SRC2.Data + (IMM*WDSZ)
```

This results in the encodings shown in table 3.3.

Arithmetic

The LARK instruction set contains only 13 arithmetic instructions, with two variants each, which cover conventional arithmetic operations, as well as comparisons and logical and bitwise operations. Such a small number of instructions are adequate because

Table 3.3: LARK Memory Instructions

Mnemonic	Encoding (Bin)	Encoding (Hex)
LOAD8U	0b01000100	0x44
LOAD16U	0b01000101	0x45
LOAD32U	0b01000110	0x46
LOAD64U	0b01000111	0x47
LOAD8I	0b01001000	0x48
LOAD16I	0b01001001	0x49
LOAD32I	0b01001010	0x4A
LOAD64I	0b01001011	0x4B
LOAD8F	0b01001100	0x4C
LOAD16F	0b01001101	0x4D
LOAD32F	0b01001110	0x4E
LOAD64F	0b01001111	0x4F
STORE8U	0b01010100	0x54
STORE16U	0b01010101	0x55
STORE32U	0b01010110	0x56
STORE64U	0b01010111	0x57
STORE8I	0b01011000	0x58
STORE16I	0b01011001	0x59
STORE32I	0b01011010	0x5A
STORE64I	0b01011011	0x5B
STORE8F	0b01011100	0x5C
STORE16F	0b01011101	0x5D
STORE32F	0b01011110	0x5E
STORE64F	0b01011111	0x5F

type distinction is achieved from the `type` and `wdsz` fields of the destination DLAR rather than the issued instruction. The only variants for arithmetic instructions is whether the instruction is scalar, operating on a single field of a DLAR, or vector, operating on an entire DLAR in parallel. All arithmetic instructions are encoded with prefix `0b10` in the high two bits, and the scalar/vector distinction is denoted by the third highest bit of the opcode, which is set to 0 for scalar operations, and 1 for vectors. This encoding is reasonably compact and leads to a human-readable prefix property. The mnemonics are written with a S or V postfix to denote if the operation is scalar or vector.

The format for arithmetic instructions is as follows.
`OP DST[DESTOFF], SRC1[OFF1], SRC2[OFF2], IMM`
Where offsets and the immediate are optional, and assumed zero if not defined.

`OP` - Opcode Field, 8 bits
`DST` - Destination LAR, 8 bits

`SRC1` - First Operand Source LAR, 8 bits
`SRC2` - Second operand source LAR, 8 bits
`OFF1` - Field offset in the first source LAR (for scalar ops), 8 bits
`OFF2` - Field offset in the second source LAR (for scalar ops), 8 bits
`DESTOFF` - field offset in the destination LAR (for scalar ops), 8 bits
`IMM` - Immediate value, 8 bits

Table 3.4: LARK Arithmetic Instruction Behaviors

Instruction	Function
`ADD`	`DST=S1+S2`
`SUB`	`DST=S1-S2`
`MUL`	`DST=S1*S2`
`DIV`	`DST=S1/S2`
`MOD`	`DST=S1%S2`
`AND`	`DST=S1&S2`
`OR`	`DST=S1|S2`
`XOR`	`DST=S1^S2`
`NEG`	`DST=~S1`
`SLL`	`DST=S1<<IMM`
`SRA`	`DST=S1>>IMM (always sign extend result)`
`SRL`	`DST=S1>>IMM`
`SLT`	`DST=S1>S2`

Table 3.5: LARK Arithmetic Instruction Encodings

Mnemonic	**Encoding (Bin)**	**Encoding (Hex)**
`ADDS/ADDV`	`0b10x00000`	`0x80/0xA0`
`SUBS/SUBV`	`0b10x00001`	`0x81/0xA1`
`MULS/MULV`	`0b10x00010`	`0x82/0xA2`
`DIVS/DIVV`	`0b10x00011`	`0x83/0xA3`
`MODS/MODV`	`0b10x00100`	`0x84/0xA4`
`ANDS/ANDV`	`0b10x00101`	`0x85/0xA5`
`ORS/ORV`	`0b10x00110`	`0x86/0xA6`
`XORS/XORV`	`0b10x00111`	`0x87/0xA7`
`NOTS/NOTV`	`0b10x01000`	`0x88/0xA8`
`SLLS/SLLV`	`0b10x01001`	`0x89/0xA9`
`SRAS/SRAV`	`0b10x01010`	`0x8A/0xAA`
`SRLS/SRLV`	`0b10x01011`	`0x8B/0xAB`
`SLTS/SLTV`	`0b10x01100`	`0x8C/0xAC`

To convert the bits of a line, rather than just change the tags, a two-operation sequence must be performed. First, a dummy load must be performed to prime a LAR with the desired type information. For an in-place conversion, load the SAME address as the source line with the desired type. Then, perform an identity operation on the line, to trigger the ALU's internal type conversion hardware.

When up-converting (ie. 8-bit to 16-bit types), values are read from the position specified by the corresponding `OFF` until the target LAR size is filled. When down-converting, or converting from an offset too far into the line to supply enough values to fill the target, results are placed starting from DESTOFF, until the number of output bits are exhausted, and the remaining fields are padded with `0`.

For vector operations, the SRC1OFF and SRC2OFF offsets are ignored for DLARs with the largest `wdsz` fields set in the operation, as they operate on an entire line of values at once. For source DLARs with `wdsz` set to d

DSTOFF determines were in the destination line the results are placed. This means the DSTOFF MUST be low enough in the line to fit the destination, and aligned to the number of values coming from the source (IE. If SRC is 32i, DST is 8u, DSTOFF must fall on a 64-place boundary. Use the OFF field in a vector op to specify which portion of a line is converted when up-converting types.

Flow Control

LARK uses only three instructions for flow control. These instructions have prefix `0x11` in the high bits. `SEL`, read "Select" is the lone branching instruction in LARK. As emitted by the compiler, the instruction encodes a LAR and offset to test for condition, and two labels. Control flow jumps to the first label if the condition is nonzero, and the second if it is zero.

`OP` - Opcode Field, 8 bits
`COND` - Condition LAR, 8 bits
`CONDOFF` - Offset into condition LAR, 8 bits
`TGT1` - Target ILAR for nonzero condition, 8 bits
`OFF1` - Field offset in the nonzero target ILAR, 8 bits
`TGT2` - Target ILAR for zero condition, 8 bits
`OFF2` - Field offset in the zero target ILAR, 8 bits
`PAD` - Pad bits, 8 bits

In the assembler, the labels are converted to the block and offset instruction format supported by the architecture. In conjunction with SLT and XOR, any common flow pattern can be implemented with SEL.

`CALL` and `RETURN` manipulate the hardware call stack described in 3.3.

Utility

In LARK there is only one utility instruction - `FETCH`. `FETCH` and is conveniently encoded as `0x00`, which simplifies bootstrapping the system. `FETCH` copies a block of

Table 3.6: LARK Flow Control Instructions

Mnemonic	**Encoding (Bin)**	**Encoding (Hex)**
`SEL`	`0b11000000`	`0xC0`
`CALL`	`0b11000001`	`0xC1`
`RETURN`	`0b11000010`	`0xC2`

instructions from memory and load into the target ILAR. At this time LARK does not include any form of instruction compression. As the benefit of this compression is less assured than other features of a LAR-based architecture, is explored elsewhere, and it's inclusion imposes significant design complexity, the current design makes no attempt to impose a compression scheme. However, because it is written with modular instructions, it would be relatively simple to replace the `fetch` instruction with one or several compressed-mode alternatives.

OP	DST	SRC1	SRC2	NUM	IMM
8	8	8	8	16	16

Figure 3.7: LARK utility instruction format

OP - Opcode, 8 bits
DEST - Destination ILAR, 8 bits
SRC1 - ILAR who's address field acts as a base address, 8 bits
SRC2 - DLAR to use for the offset, 8 bits
NUM - Number of contiguous ILARs to be loaded, 16 bits
IMMEDIATE - Immediate value for address calculation, 16 bits
Addressing is again performed by the rather odd:
`Address=SRC1.Address+Src2.Data+Immediate`

This operation is, however, not generated by the compiler. Instead, it is inserted as part of the packing and alignment process in during assembly.

Table 3.7: LARK Utility Instruction

Mnemonic	**Encoding (Bin)**	**Encoding (Hex)**
`FETCH`	`0b00000000`	`0x00`

Calling Convention, Persistent Pointers

In order to execute useful code, a set of calling conventions must be established. These conventions are to meet similar requirements to any architecture; provide ac-

cess to constants, ensure orderly transitions between functions, and generally provide guidelines for the use of shared resources. First, and similar to several established architectures like MIPS, DLAR0 is a constant 0. All of the native number encodings understand a field full of logic 0 as `0` in their number system, and the ALU features type conversion hardware, so the 0 register is inherently polymorphic.

DLAR1 is kept tagged as a 64bit unsigned integer, to hold vector of important, frequently accessed values. These include the following pointers:
`CP` - Points to the beginning of the current constant pool.
`SP` - Points to the top of a stack of return blocks (one LAR, supports offsets)
`FP` - Frame pointer, Points to a DLAR (Possibly more than one) containing the current context.

Table 3.8: DLAR1 Contents

0	1	2	3	4	5	6	7
CP	SP	FP					

This document does not establish exactly how the stack will work; because of the same problem as packing data of differing types in general, no overwhelmingly satisfactory solution has been discovered, straightforward possibilities either burn an excessive number of DLAR names to map a contiguous memory used for a set of variously-typed data, or require complicated indirected addressing schemes to separate data in differing types.

Bootstrapping a machine with ILARs is not significantly more difficult than a conventional architecture. In particular for LARK, because the `FETCH` opcode is `00`, placing a single ILAR-sized chunk of instructions at the bottom of the memory space and executing an all-0 instruction at startup will load the block from the bottom of memory into ILAR 0, which gives 8 initial instructions to `FETCH` some bring-up code and `SEL` to it.

Examples

Beginning with a single-instruction example, figure 3.8 shows the behavior of a typical load operation. Specifically, figure 3.8 shows the critical parts of the DLAR file during the execution of the instruction `LOAD16I D3 D0 D5 0`. This instruction specifies that DLAR `D3` be type-tagged for 16-bit signed integers, and loads the 2048-bit-aligned block containing the memory specified by the other operands. Note the addressing mode allows for a datum in an already-loaded LAR to be used as an absolute pointer by setting SRC1 to `D0`, specifying the DLAR whose offset is currently pointed at the desired address as SRC2, and leaving the immediate value as `0`. Because it is an aligned load, though the base address is two bytes into the range, the base address is written to the address field, the data is copied on aligned boundaries, and the low bits are interpreted as an offset in terms of `WDSZ`-sized items, stored in the offset field;

in this case, 1. This example shows a byte-addressed memory with 16-bit addresses for brevity, though LARK is specified with a 64-bit address space.

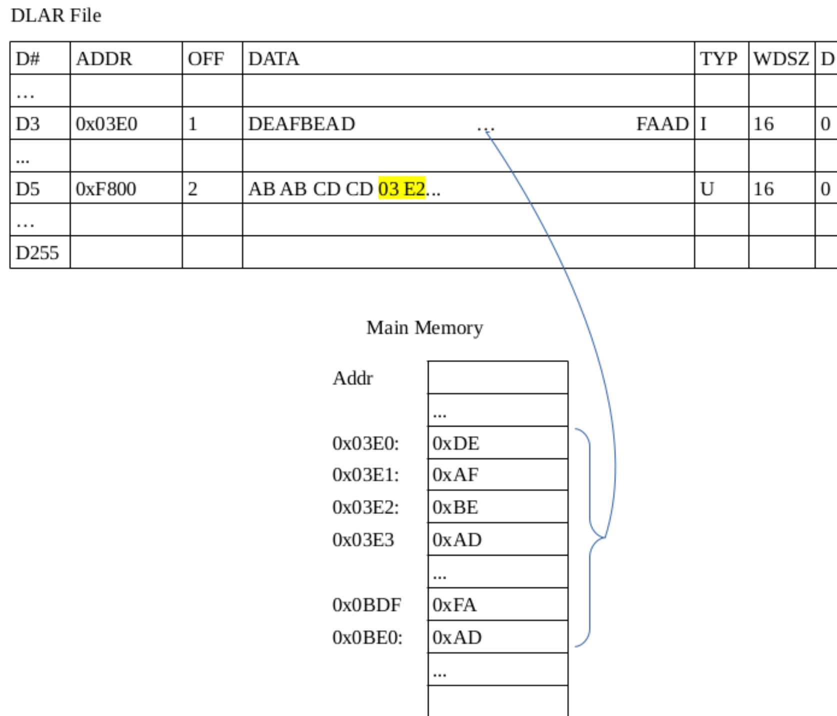

Figure 3.8: Behavior of `LOAD16I D3 D0 D5 0`

Figure 3.9 illustrates a load of an already-cached segment of memory. This instruction would cause no additional memory bus traffic. Instead, it will tag the target DLAR with the type and wordsize specified in the instruciton and same address as an already-in-use DLAR, and alter the DLAR file map to point the target DLAR's data section (`D6`) to the *same* piece of memory already mapped for that base address (in this case, `D3`) – thus providing two differently-annotated handles on the same line of data. Also note that here the addressing scheme is used to take the base address of another DLAR plus an immediate offset by specifying a DLAR holding the desired base address as the first source operand, `D0` as the second source operand, and an immediate value.

To demonstrate one of the most confusing aspects of LARK, figure 3.10 shows the a copy being performed with a sequence of instructions ending in a store. This is performed in several steps to first set up the addresses, then associatively copy the desired values into a DLAR to re-tag with an aliased load as in the previous example, and finally to issue a store to change the address tag of the copy. The first operation is once again treating `D5` as a line containing pointers, set its offset appropriately to the desired target address before this store with a `LOAD D5 D5 D0 3` instruction. The copy is much like the previous example, in this case a `LOADF32 D9 D3 D0 0` to prime `D9` as the location for the copy. Finally, issuing a `STORE16U D9 D0`

D#	ADDR	OFF	DATA	TYP	WDSZ	D
…						
D3	0x03E0	1	DEAFBEAD … FAAD	I	16	0
...						
D6	0x03E0	2		F	32	0
…						
D255						

Figure 3.9: Behavior of subsequent `LOADF32 D6 D3 D0 2`

`D5 0` will set DLAR `D9`'s address to the address indicated by the arguments, the type tags to the values indicated in the instruction, and *copy* the data in source DLAR (`D3`) to the storage of the destination DLAR (`D9`), breaking the associativity. This operation will set the dirty bit on the destination line `D9` to mark it for write-back to main memory in the same way as any other operation requiring main memory be eventually updated due to a change in the stored data.

D#	ADDR	OFF	DATA	TYP	WDSZ	D
…						
D3	0x03E0	1	DEAFBEAD … FAAD	I	16	0
...						
D5	0xF800	3	AB AB CD CD 03 E2 04 E6			
...						
D9	0x04E0	3	DEAFBEAD … FAAD	U	16	1
…						
D255						

Figure 3.10: Performing a copy with LARs

For a larger demonstration, the below describes a a trivial example program as pseudo-C, with the body compiled to both LARK and MIPS-like assembly. Remember, like a conventional register, LARs are able to hold values being operated on. Like a cache line, LARs can contain a number of spatially proximal scalar values. As in SWAR, a LAR is able to hold a vector of values to be operated on in parallel. Like their progenitor CRegs [6] LARs are able to transparently resolve ambiguous aliases in hardware.

Which yields the following counts:

The reason there are ranges on the LARs memory access counts is because if the used locations are aliased to any "live" value, the memory access is replaced with a

```
nasty(int* i, int* j, int* k)
{
    i=j+k;
    k=j&k;
}
```

LARs	MIPS
	LW $t1, j($sp)
	LW $t2, 0($t1)
LOADSW D1 D31 0 j	LW $t3, k($sp)
LOADSW D2 0 D1 0	LW $t4, 0($t3)
LOADSW D3 D31 0 k	LW $t5, k($sp)
LOADSW D4 0 D3 0	LW $t6, 0($t4)
LOADSW D5 D31 0 i	ADD $t6, $t2, $t4
LOADSW D6 0 D5 0	SW $t6, 0($t5)
ADDS D6 D2 D4	LW $t2, 0($t1)
ANDS D4 D2 D4	LW $t4, 0($t3)
	AND $t4, $t2, $t4
	SW $t4, 0($t3)

Figure 3.11: Sample code compiled for MIPS and LARK

Table 3.9: Instruction Count Comparison for Sample Code

	LARs	MIPS
Memory Accesses	0-5	9
Reads	0-3	7
Writes	0-2	2
Total Instruction Count	9	11

simple associative update. Even if we allow that the MIPS version may have passed it's parameters in registers, it would only reduce the number of instructions to parity at eight, and there would still be eight assured memory accesses for the MIPS version. While it is true that some or all of these memory accesses may be satisfied from cache, this would still require traffic across the memory interface. An associative update in the LARs version is entirely internal to the processor, and does not incur any bus traffic. It is also worth noting that the LARs version could operate on entire vectors, each the length of the data field, by only changing the ADD and AND operations to their vector forms. There would be no additional memory references nor additional cycles in processing the operations.

In an ideal case, a LAR based architecture as currently conceived would be able to execute nearly 128 operations per cycle, presuming a predictably ordered instruction stream, and 8-bit data which comes in at least LAR-width vectors. A task with such a high degree of regularity and parallelism is extremely unlikely outside of certain highly specialized multimedia applications. A more realistic estimate will still provide excellent execution density, for several reasons. Firstly, because of the inherently

parallel nature of LAR loads, and complete absence of explicit stores, there will be a large savings on memory access instructions. Likewise, where ILP is available it can be easily and transparently exploited with exactly the same mechanisms used for scalar operations, with no mode changes as are inherent to many SWAR designs [7]. Finally, the LAR's use of dirty tags and alias analysis precipitously reduce the number of loads from main memory, preventing stalls while waiting for main memory.

3.4 The Simulator

In order to explore a full-datatype LAR-based system, a simple simulator matching the LARK spec has been implemented.

The simulator, LARKem (LARK-Emulator), is written in in standard C, with a sprawling collection of macros to hide the relatively complicated struct used to represent each DLAR, and associated selection logic to perform typed operations. Like LARK itself, LARKem is designed to be as simple as possible to manipulate, rather than performance or accuracy, and it has many internal decisions that reflect this bias. The source code for LARKem is attached as appendices beginning with A.1.

In LARKem, each LARK instruction is implemented as a C function, to keep their behavior as orthogonal as possible. However, because the bulk of each instruction is accessor boilerplate, each function is actually generated by a two-layer deep system of macros. The outer layer of macro is the accessor logic for performing the operation `aluops.h`, `memops.h` These require several odd sub-classes to account for typing rules. For example, the macros `SALUBINOP` and `VALUBINOP` in `aluops.h` are a macros (with some helpers of their own, no less) which handles the necessary selection logic and casting to perform various bitwise operations on types that C's type system don't allow bitwise operations on. These macros are then called in `LARKemNoSIMD.c` to implement most of the machine operations, with occasional exceptions (such the unary `not` operators). Each of these operand type macros in turn relies on the massive DLAR accessor macro found in `DLARConvert.h`

One decision which makes the included version slightly incorrect but much simpler is the use of a full host `float` for all of the narrower floating-point types. While it would be possible to generically implement the smaller floats not supported by host systems using code like that found in 3.1, the additional twiddling was deemed not worth it, particularly because a faster, more precise simulator would most likely handle at least half-precision (16-bit) floats with non-portable extensions, like compiler intrinsics to use the half-precision conversion support offered by the `F16C` instruction added to x86 with AVX [2].

```
//For 16 bit floats
i=(t.v.i >> (52-10) & 0x3ff)
e=(((((t.v.i >> 52) - 1023) & 0x1f) + 15) << 10)
if(t.v.i<0)
        {s |= 0x8000)}
```

```
//For 8 bit floats
        // sign1:exponent4:mantissa3, exponent bias of 7.
mantissa=(t.v.i >> (52-3) & 0x7)
exponent=(((((t.v.i >> 52) - 1023) & 0x0f) + 7) << 3)
if(t.v.i<0)
        {s |= 0x80)}
```

Listing 3.1: Sample code for implementing non-standard narrow floats

One major enhancement that has not been attempted in the current version of the simulator is the use of SIMD intrinsics to substantially accelerate the vector operations. As most modern architectures offer some manner of SIMD extension, it would be possible, with considerable effort, to map at least a subset of LAR vector operations onto the hardware SIMD support. Because no platform's selection of SIMD-enabled operations will neatly match LARK's needs, and no two platform's selection of supported operations will be exactly the same, the effort to write a high-performance simulator for a proof-of-concept architecture doesn't seem worthwhile. The effort could be constrained somewhat with the use of compiler intrinsics, but this would result in far more complicated, less portable, and less adaptable code, defeating the main purpose of LARKem as an easy-to-adapt research tool.

Copyright© Paul Selegue Eberhart, 2019.

Chapter 4 The Compiler

In the initial project plan, the objective was to produce an LLVM [32] backend which could generate code for the LARK spec. LLVM, initially standing for "Low Level Virtual Machine," but since officially stripped of it's acronym connotation, is a collection of modular compiler and toolchain technologies. LLVM was initially viewed as the most promising existing infrastructure by reputation, and early explorations into features of the tooling and intermediate representation. By the same token building on top of GCC was rejected very early in the process due to the extreme learning curve to make substantive changes to it's code base [33], and lack of generic tooling for handling multi-width registers.

However, the state of the LLVM code-base and documentation at the time (2011-2012) scuttled that idea. The documentation referred to the MIPS back-end as an example, and after some exploration it was determined that the MIPS back-end did not even generate valid code, and the model it was built on was completely incapable of supporting the desired memory behavior. The single largest disincentive to attempting to use LLVM is that it promoted almost all data types to a generic virtual machine-word in an early pass, making the source-to-hardware width tracking required to effectively allocate LARs impossible within the existing infrastructure. Another major problem is that, while the TableGen tooling used to define a target architecture in LLVM [34] includes a concept of `SubRegs` which initially looked applicable to LARs, it did not appear to actually support structures as complex as a LAR. Finally, the rate of code-churn on the interfaces in LLVM at the time (2011-2012) was so severe that by the time the author was up to speed to evaluate the relevant internal structures and behaviors, they had almost all been significantly changed upstream in the new release in only a matter of months.

Instead, constructing a minimal reference design was selected as a more promising approach. Thus, the special-purpose language, termed LARC as it is a C-like construction designed to target LARs, and is merely a proof-of-concept design (named from the expression "On a lark"). LARC includes a number of simplifying additions, chiefly around adding language support for native vectors, allowing explicit use of architectural features without having to include still-difficult analysis features like automatic vectorization. The LARC frontend has been implemented with pccts/Antlr 2 [35].

Obviously an ideal toolchain for LARs would perform automatic vectorization and packing into LAR-sized units. Based on the difficulty present in effective automatic vectorization and packing for SIMD architectures, performing the far more general case of the similar problem for LARs fits in a semantic niche somewhere in the vicinity of "wanting a unicorn pony." The persistent difficulty of performing automatic vectorization appears to be an enduring killer of over-ambitious computer architectures.

4.1 Input Language

A straw-man language, LARC, has been designed to illustrate the behavior and information tracking suitable for targeting a LAR-based design. The LARC input language is a C-like language, but is in no way an exact super- or sub- set of ANSI C. Rather, a number of C language features have been omitted for ease of implementation, and a small number of features to ease the demonstration of LAR features have been added.

LARC's type system is not extensible as in C - there are no user-defined data types - but is initially much larger. LARC supports C99-style types for all the native types supported by LARK - `uint8_t, uint16_t, uint32_t`, as well as first-class native-width vectors of all the types, denoted with an infix "vec" in the identifier name, creating types like `uint8vec_t`, etc.

A grammar for the language has been specified as an ANTLR2 [36] compatible grammar. ANTLR is a highly flexible parser-generator which accepts $LL(k)$ grammars in a specialized EBNF (extended Backus–Naur form) format, and is capable of automatically generating an abstract syntax tree and stub code for working with parses of that grammar. `larcast.g` in the appendix is the larc grammar, annotated appropriately to build an abstract syntax tree generator.

This LARC has no relation to the Univac LARC (Livermore Advanced Research Computer) of the early 1960s.

4.2 Register Allocation and Vectorization

A primary interesting difference between compiling for an architecture based on LARs and compiling for a conventional register-based architecture is the problem of register allocation. Modern architectures uniformly include a relatively small set of independent registers, possibly with some overlap or aliasing, such as the reuse of the x87 floating point registers for MMX instructions, and no direct relationship to memory image.

LARs are comprised of a relatively large memory, mapped 1:1 with main memory, and have complicated rules about acceptable packing. This has a variety of implications for the allocation problem. First, it convolutes the problems of memory layout and register allocation; because LARs are 1:1 mapped with main memory, allocating variables to LARs determines both the in-memory structure and the register allocation.

A naive approach is to allocate DLAR, with that DLAR kept loaded across the liveness of all the variables stored in the memory range corresponds to. This means that, for scalar code, one strategy is to use a particular DLAR-sized block of memory will be used to store like-typed variables whose liveness interval overlaps as much as possible. Scalar operations can then be readily performed between loaded DLARS, or even between different offsets in the same DLAR, with only the minor complication of guaranteeing offsets be aligned with the size of the type in question.

In more detailed description of this naive approach would be:

- For the first variable of a type seen in the current block being analyzed, select an unused memory region, mark it with the appropriate type, and place the value in it
- for subsequent values of the same type, load them consecutively into the open DLAR for that type
- when the current DLAR for the type being allocated is full, allocate another one at the next available location

with a numbe of relatively low-hanging variations possible optimizations, such as

- separating allocations of arrays from scalars so that arrays begin on a LAR-size aligned boundary to maximize opportunities for natural vector alignment
- growing the different-type regions from widely separated base addresses, such that long memory regions to encourage natural packing
- analysis around `struct` like data structures to ensure that like-members are serialized into like-offsets in a set of DLAR-sized chunks to maximize the likelihood of trivially extractable parallelism

Unfortunately, this analysis has to be performed on relatively large units. The traditional approach of performing initial analysis on basic blocks will not work well within this regime; in fact, this will work better the larger the block it is performed over, as it will maximize both memory packing/DLAR utilization an expose maximum parallelism.

Alternatively, at a cost of occupying more DLARs for the same data, it is possible to issue a series of `STORE`s with the same base address but different type tags to different DLArs, loading them with differently type-tagged views of the same memory. This makes the alignment problem *much* harder as it involves the packing of different-sized values such that the alignment constraints for each type are respected. This is, however, extremely costly in terms of DLAR pressure (it occupies a DLAR name for each type the line must be accessed as), prevents the use of vector instructions on those data, and leaves an avenue for accidental operation on bits as though they are the wrong type via offset errors. This seems less preferable, but bears mention that, because of their tagged direct-mapping to memory, LARs should be relatively flexible in their use, particularly when compared to the long registers in SWAR/SIMD systems, which can typically only be used when the data has been carefully pre-massaged into alignment.

On the topic of similarity to SWAR designs, DLARs are also inherently vectors, which brings in the relatively well-known vectorization problems to efficiently utilizing a DLAR-based machine. The fact that they are type-tagged is not an *additional* complication for vector operations, as the requirement that that like-typed data be packed into a single vector is already present, and the address tags make the problem slightly easier, as there will be no requirement to flush painstakingly-aligned data due to potential aliasing. For vector operations, the LAR allocation problem is also

difficult, but in a somewhat more precedented way. It is largely analogous to the problem of loading (packing) the vector registers on processors with SIMD extensions, eg. packing a set of 8 16 bit values into a 128bit XMM register for SSE2 such that packed operations can be performed on all eight values simultaneously.

Attempting to *automatically* vectorize by packing vector-sized sets of scalar operations is an active area of research for production compilers. Speaking in generalities, these techniques tend to not be particularly effective as compared to hand-coded vector support written with purpose-made data types and machine-specific compiler intrinsics. However, even small gains in exploiting parallelism have potential for large improvements in performance, and hand-coded vector operations are both difficult to write and machine specific, so the premise of even mildly effective automatic vectorization is extremely attractive.

Some transformations, like loop vectorization in which loops with no (or analyzable) data dependencies are unrolled into machine-vector-sized blocks to be performed in parallel, are relatively well-understood, but only applicable in narrow circumstances. Other more general transformations, like those intended to extract superword-level parallelism by locating indipendent scalar operations that happen to require similar sequences of instructions on same-type data [37] require very large-scale analyses, typically crossing basic blocks in order to accrue collections of similar operations of a size that makes vectorization practical.

The currently (on the scale of decades) in-vogue preference toward graph-coloring register allocation [38] is wildly suitable for LAR allocation for a variety of reasons, from it's lack of facility for tracking or favoring spatial locality, to the basic restriction that the NP hard coloring problem does not tolerate performing allocations on large blocks of code, as required when performing vectorization, well. However, a greedy algorithm will tend to favor spacial locality, better tolerates large blocks due to lower computational demands, and is easier to adapt for allocations partitioned by type. A linear-scan register allocation scheme [39], as described by Poletto and Sarkar, modified for LARs, would be very similar to the naive LAR allocation approach described above, and this similarity lends both legitimacy and an existing body of work from which some optimizations and refinements may be borrowed.

Spilling

Spilling is the act of evicting less-critical data from the obligatorily small upper levels of a memory hierarchy to make room for more urgently needed data. In most conventional architectures, spilling is performed on the register file by storing values currently in the separately-addressed registers back to main memory. Register spilling is a notably different issue in LAR-based designs, as the entire top portion of the memory hierarchy is explicitly, statically software managed, and all data promoted in the memory hierarchy are origin tagged within the same addressing scheme. In principle, the but can be accomplished with more sophistication and finesse because the explicitly compiler-managed nature of LARs renders LAR-spilling a compile-time problem. A downside of this is that, like all compiler-managed memory models,

platform differences to the memory hierarchy require recompilation or changes to run-time systems, where transparent caches provide opportunistic improvement.

This also creates a significant obligation on the operating system and calling conventions for LAR-based deigns to ensure that, at unit boundaries like function calls or program scheduling events, the proper LARs are (re)loaded with the proper data.

4.3 The Assembler

Assembling for LARK, like any design using ILARs, makes demands on the assembler that no conventional architecture would. Early in this project, the assembler was declared explicitly out of bounds, however, discussing some of the reasons assembling for ILARs is a challenge is worthwhile. The primary source of difficulty is that ILARs must be explicitly `FETCH`ed such that the instruction block will be in the appropriate ILAR before it can possibly be reached, and `FETCH` is an instruction itself, the first obvious challenge is that `FETCH`es must be inserted into the instruction stream in proper locations such that all possible instruction targets will always be loaded before they can be reached. This breaks down into several serious sub-problems. One obvious issue is that the assembler must be able to accurately track all reachable code paths so that it can make the basic guarantee that needed instructions will be fetched into an ILAR early enough to ensure they are loaded by the time they are reached. Additionally, because the ILAR file is of limited size, the assembler needs to keep track of instruction reachability. This process is similar to liveness analysis on variables; reachable instructions must be kept in ILARs, and ILARs no longer containing reachable instructions need to be tracked for reuse. Finally, the insertion of the `FETCH`es adds instructions to the program to be scheduled, so the process of inserting fetches changes the scheduling problem as it proceeds.

There are some fallback options that would allow relaxation of these demands on the assembler. Unlikely code paths could be left in main memory, and calls to them indirected through loader routine which will determine and preform the appropriate `FETCH`es before returning control. Similarly, some fraction of ILAR slots could be reserved for inserting `FETCH`es or loading and redirecting flow into additional ILAR sized blocks of fetches or code for computing fetches.

In order to check for consistency, an AIK [40] specification which can assemble LARK instructions has been written. This assembler is not adequate to generate usable code without assembly with pre-inserted fetches, as from hand-written code or a compiler capable of performing fetch scheduling and insertion on its own.

4.4 Virtual Memory

While designing a virtual memory system for a LAR-based architecture is far out of scope for this work, considering what it would entail is illustrative to the alterations to memory behavior required to utilize LARs.

Much like in a system with a conventional memory hierarchy; leveraging a virtual memory system would allow for, among other desirable behaviors, isolation and

relocation of binaries, in exchange for some added complexity in both hardware and required linker and OS support. The first decision that would be required for virtual memory on LARs is whether the LAR-file address tags would be in terms of logical of physical addresses- whether the LAR tags pass through the MMU. In a VM system which uses virtually-tagged LARs would have the problem that dual-mapped segments would no longer be guaranteed alias-free unless additional checking were performed in the virtual memory system itself. In a VM system which uses physically-tagged LARs, every operation that mutates the address field of a LAR must go through the MMU which even accelerated by a TLB (Translation Lookaside Buffer) could potentially impose an unacceptable performance penalty on routine operations.

Copyright© Paul Selegue Eberhart, 2019.

Chapter 5 Results

The primary result of this work is the determination that LARs represent a sufficient disruption to the status-quo that there is not a ready incremental adaptation of existing technologies to work for or with a LAR-based design. Because LARs' tagging convolves memory layout and register allocation, there is not a straightforward way to adapt existing register allocation techniques to LARs. Conversely, the need for statically predictable memory behavior makes it impractical to bolt LARs into a design with a conventional cached memory hierarchy.

In the path to make this determination, a variety of artifacts have been produced.

First, LARK, a fleshed-out ISA based entirely on LARs was designed as a demonstrator for the purposes of evaluation. The LARK specification is complete enough to implement, though certain supporting decisions about usage conventions, particularly stack behavior, have not been made for lack of evidence. A software simulator, LARKem, was written to verify this design and act as a target for testing the software toolchain. LARKem has never functioned fully, for lack of a front-end, but the code for the basic data structures and opcodes has been written and unit tested.

Secondly, some preliminary attempts at a software toolchain have been undertaken. After adapting an existing toolchain was determined impractical, the beginnings of a full-custom toolchain have been designed and written. First, LARC, a C-like language has been designed to address the particular needs of LARs in terms of type system and native vectors. This design was translated into an ANTLR grammar, which has been annotated to generate an abstract syntax tree, onto which tree-walkers can be attached to test algorithms for suitability. Some preliminary algorithms for performing LAR allocations are suggested.

Similarly, a grammar for generating machine code from assembly has been written. This grammar is not particularly useful as an assembler, as it is not capable of scheduling and inserting data movement instructions, as would be required in a practical LARK toolchain.

Finally, a far reaching literature survey situates LARs in the larger computing context. This survey reveals a number of curious comparisons with historical designs, reveals interesting trends in prior efforts to address the VonNeumann bottleneck.

Copyright© Paul Selegue Eberhart, 2019.

Chapter 6 Conclusion

If you tilt your head to the right, LARs look like the union of two resounding architectural successes; SIMD and clever scheduling to reduce stalls due to memory traffic. If, however, you tilt your head to the left, it looks like the union of two resounding failures; exposed VLIW and software-managed memory hierarchies. This position between a "good" and "bad" idea makes LARs an interesting study in the conditions and that set those technologies up for their respective successes and failures.

Perhaps the most interesting thing to come out of this work are the notes on various long-running trends in computer architecture, particularly a 25-year slide toward extremely dynamic memory behavior, and . LARs represent another path, moving back towards relatively flat, static, and extremely predictable memory behavior, scheduled cleverly by the compiler and/or programmer, rather than embracing dynamism outside the reasonable comprehension of any human.

One of the thing made evident by the historical survey is that many of the antecedent technologies for LARs went out of fashion largely because of the difficulties they present for high-level languages and dynamic, multitasking environments. Multitasking environments and their confounding effects on static scheduling have lead to a a dramatic move toward dynamically scheduled systems whose actual execution and memory semantics are largely invisible to the programmer and compiler. Furthermore, as memory access times become non-deterministic with more sophisticated memory devices and hierarchies, attempting to statically schedule around memory delays even at the micro-level becomes an impossible exercise in bounded conservatism and contingency plans between each step of any non-streaming memory read. Likewise, most widely-used languages have, at best, a handful of types, which are typically poorly defined in terms of bit width (In C, a `short` is no longer than an `int`...), and at worst rely entirely on type inference, making the effective utilization of constrained, narrow types difficult. These are reasonable design decisions for languages intended to target machines with a clear native machine width, but create a serious impediment to effectively mapping to devices with multiple supported encodings and precisions, like SIMDs or LARs. There has been some motion on this front, like the addition of the `<stdint.h>` types in C99, and analogous more-specific types present in other recent systems languages like Rust and Go, as well as various domain specific languages targeting GPUs, SIMD extensions, or specialized scientific computing applications. These developments are an encouraging indication of a recognized need.

In another move toward LAR-friendly design spaces, in the last few years we have started to observe cracks in the edifice of dynamic optimization with caches and multi-issue pipelines hidden entirely below the ISA abstraction that typifies modern computer architectures. The extreme complexity of these designs forces compiler writers to perform a different set of acrobatics to model various implementation details, and the dynamic rescheduling devices themselves have a number of drawbacks.

One drawback lies in the power, and far less importantly, chip area used by the collection of always-active circuitry involved in instruction decoding, cache management, and branch prediction hardware. Another recent and extremely high-profile

Speculation attacks which exploit unexpected statically unpredictable but dynamically observable behaviors of these hidden-behind-the-ISA optimizations have become a significant threat [41], providing a new impetus to consider the value of a return to more static behavior.

A disadvantage of turning away from dynamic designs is that static architectures inherently expose lower-level details to the programming interface, hobbling portability. In the modern era of proprietary software and unreasonable build times even for source-available software, the most viable alternative is binary translation, using some kind of software shim to just-in-time, statically, or iteratively translate from a distribution binary into a machine-specific form. This method was attempted by Transmeta, who employed "code morphing," an iterative translator, to convert x86 binaries to run on their VLIW designs [42]. Whether they were eventually out-competed by designs that did their translation dynamically in microcode *because* of their choice to do their translation at a slightly higher level, or this was incidental is not clear, and the continued success of "microcode" driven binary translationm, from the IBM System/38's machine interface (intermediate language) in the late 1970s [17] to the micro- and macro- op fusion in Intel [43] x86 parts since 2011 implies this option will continue to be an option to separate ISA-as-API and implementation.

LARs are expressly designed to address the modern reality that parallelism is required for performance, and scalar operations are required for reasonable programming models, so intermixing the two should be natural and with little penalty. While they appear very foreign when compared to recent computer architectures, and place different demands , LARs are an avenue to exploring areas of the computer architecture design space long ignored, sometimes for contextually-appropriate reasons, in the light of recent developments in the rest of computing. By rearranging the choices on which behaviors are exposed in the programming model, LAR-based designs provide an opportunity to move toward simpler, more predictable, and more efficient designs that make better use of hardware resources, while also presenting a simpler model to compiler writers. The design of the LARK ISA, historical situation of LARs among extant architectures, and documentation of the toolchain requirements to target LARK in this work serve to explore and demonstrate the viability of LAR-like designs as a promising direction for computer design.

Copyright© Paul Selegue Eberhart, 2019.

Appendix A LARKem

Listing A.1: LARKemNoSIMD.c

```
#include "LARKemNoSIMD.h"
#include "DLARConvert.h"

//FIXME: I need to police the DLAR[0] =0 and DLAR[1]= [PC,CP,SP,FP] as
   UINTVEC64X32 restrictions somehow.
//FIXME: Change the casts for bitwise operations on floats to forcably
   touch the wrong element of the union.


//Stuff for reading in assembly
char asmline[ASMBUFSZ];
FILE * asmFile;

//Declare the data LAR file
DLAR_t DLARFile[256];

//and the instruction LAR file
ILAR_t ILARFile[256];

//AAAND a RAM... (Tiny for now)
uint8_t RAMFile[MEMSZ];


//DLAR0 is 0
//DLAR1 starts out [CP, SP, FP,...] and is used for status
//Do I need a Block:Offset PC in there?


//Temporaries for ALU
//vector path
DLAR_t ALULARA;
DLAR_t ALULARB;
//scalar path
scalarbuf_t ALURegA;
scalarbuf_t ALURegB;

//Quick-and-dirty main
//      This needs to be replaced with something to use assembled code
int main(int argc, char* argv[])
{
        //DLAR0 is polymorphic 0, always.
        //I'm not sure if there is a convienent way to make it static
           and addres the same as others...
        DLARFile[0].addr=0;
        DLARFile[0].flags=0x00;
        int x;
        for(x=0;x<255;x++)
```

```
        {
                DLARFile[0].data.UINTVEC8X256[x]=0;
        }
        if(argc > 0)
        {
                asmFile=fopen(argv[1],"r");
        }
        else
        {printf("Usage: larkem asmfile.lasm");}
        if(asmFile==NULL)
        {printf("Invalid asmfile/file not found");}

        //Read lines until end of file
//      while(fgets(asmline,ASMBUFSZ,asmFile) != NULL)
//      {
//              instrparse();
//      }
//      return 0;
 }
//
//
// //Parse out a line of assembly
// int instrparse()
// {
//      int items = 0;
//      char *item;
//      do
//      {
//              item=strtok(asmline, " ");
//
//              items++;
//      }while(item!=NULL);
//      return 0;
// }

/*
//Decode that instruction:
int instrdec()
{

        switch(opname)

        "LOAD8U"
        "LOAD16U"
        "LOAD32U"
        "LOAD64U"
        "LOAD8I"
        "LOAD16I"
        "LOAD32I"
        "LOAD64I"
        "LOAD8F"
        "LOAD16F"
        "LOAD32F"
        "LOAD64F"
```

```
        "ADDs"
        "SUBs"
        "MULs"
        "DIVs"
        "MODs"

        "ANDs"
        "ORs"
        "XORs"
        "NEG"
        "SLL"
        "SRA"
        "SRL"

        "SLT"


        "SEL"
        "CALL"
        "RETURN"
}

*/
//Arithmetic Instructions: Scalar Mode

SALUOP(ADD, +, dst, src1, src2, off1, off2, destoff, imm)

SALUOP(SUB, -, dst, src1, src2, off1, off2, destoff, imm)

SALUOP(MUL, *, dst, src1, src2, off1, off2, destoff, imm)

SALUOP(DIV, /, dst, src1, src2, off1, off2, destoff, imm)

SALUBINOP(MOD, %, dst, src1, src2, off1, off2, destoff, imm)

SALUBINOP(AND, &, dst, src1, src2, off1, off2, destoff, imm)

SALUBINOP(OR, |, dst, src1, src2, off1, off2, destoff, imm)

SALUBINOP(XOR, ^, dst, src1, src2, off1, off2, destoff, imm)

int doNOTs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, uint8_t imm)
{
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if((typeconverts(&ALURegA, dst , src1, off1) == 0) && (
           typeconverts(&ALURegB, dst , src2, off2) == 0))
        {
                switch(dsize)
                {
```

```
case 0x00: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC8X256[destoff] = ~(ALURegA.
            data.UINT8); break;
        case 0x20: DLARFile[dst].data.
            INTVEC8X256[destoff] = ~(ALURegA.
            data.INT8); break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC8X256[destoff] = (float8_t)
            (~((int64_t)ALURegA.data.FLOAT8));
            break;
} break;
case 0x40: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC16X128[destoff] = ~(ALURegA.
            data.UINT16); break;
        case 0x20: DLARFile[dst].data.
            INTVEC16X128[destoff] = ~(ALURegA.
            data.INT16); break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC16X128[destoff] = (float16_t
            )(~((int64_t)ALURegA.data.FLOAT16));
             break;
} break;
case 0x80: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC64X32[destoff] = ~(ALURegA.
            data.UINT32); break;
        case 0x20: DLARFile[dst].data.
            INTVEC32X64[destoff] = ~(ALURegA.
            data.INT32); break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC32X64[destoff] = (float32_t)
            (~((int64_t)ALURegA.data.FLOAT32));
            break;
} break;
case 0xC0: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC64X32[destoff] = ~(ALURegA.
            data.UINT64); break;
        case 0x20: DLARFile[dst].data.
            INTVEC64X32[destoff] = ~(ALURegA.
            data.INT64); break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC64X32[destoff] = (float64_t)
            ~(((int64_t)ALURegA.data.FLOAT64));
            break;
} break;
        }
}
```

```
        else
        {return 1;}
}

int doSLLs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, uint8_t imm)
{
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if ((typeconverts(&ALURegA, dst , src1, off1) == 0) && (
           typeconverts(&ALURegB, dst , src2, off2) == 0))
        {
                switch(dsize)
                {
                        case 0x00: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                   UINTVEC8X256[destoff] = (ALURegA.
                                   data.UINT8) << imm; break;
                                case 0x20: DLARFile[dst].data.
                                   INTVEC8X256[destoff] = (ALURegA.data
                                   .INT8) << imm; break;
                                case 0x30: DLARFile[dst].data.
                                   FLOATVEC8X256[destoff] = (float8_t)
                                   ((int64_t)(ALURegA.data.FLOAT8) <<
                                   imm); break;
                        } break;
                        case 0x40: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                   UINTVEC16X128[destoff] = (ALURegA.
                                   data.UINT16) << imm; break;
                                case 0x20: DLARFile[dst].data.
                                   INTVEC16X128[destoff] = (ALURegA.
                                   data.INT16) << imm; break;
                                case 0x30: DLARFile[dst].data.
                                   FLOATVEC16X128[destoff] = (float16_t
                                   )((int64_t)(ALURegA.data.FLOAT16) <<
                                    imm); break;
                        } break;
                        case 0x80: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                   UINTVEC64X32[destoff] = (ALURegA.
                                   data.UINT32) << imm; break;
                                case 0x20: DLARFile[dst].data.
                                   INTVEC32X64[destoff] = (ALURegA.data
                                   .INT32) << imm; break;
                                case 0x30: DLARFile[dst].data.
                                   FLOATVEC32X64[destoff] = (float32_t)
                                   ((int64_t)(ALURegA.data.FLOAT32) <<
                                   imm); break;
                        } break;
```

```
                    case 0xC0: switch(dtype)
                    {
                            case 0x10: DLARFile[dst].data.
                                UINTVEC64X32[destoff] = (ALURegA.
                                data.UINT64) << imm; break;
                            case 0x20: DLARFile[dst].data.
                                INTVEC64X32[destoff] = (ALURegA.data
                                .INT64) << imm; break;
                            case 0x30: DLARFile[dst].data.
                                FLOATVEC64X32[destoff] = (float64_t)
                                ((int64_t)(ALURegA.data.FLOAT64) <<
                                imm); break;
                    } break;
            }
    }
    else
    {return 1;}
}



//FIXME: C's right shifts are implementaiton defined...
int doSRLs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if ((typeconverts(&ALURegA, dst , src1, off1) == 0) && (
            typeconverts(&ALURegB, dst , src2, off2) == 0))
        {
                switch(dsize)
                {
                        case 0x00: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC8X256[destoff] = (ALURegA.
                                    data.UINT8) >> imm; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC8X256[destoff] = (ALURegA.data
                                    .INT8) >> imm; break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC8X256[destoff] = (float8_t)
                                    ((int64_t)(ALURegA.data.FLOAT8) >>
                                    imm); break;
                        } break;
                        case 0x40: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC16X128[destoff] = (ALURegA.
                                    data.UINT16) >> imm; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC16X128[destoff] = (ALURegA.
                                    data.INT16) >> imm; break;
```

```
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC16X128[destoff] = (float16_t
                                    )((int64_t)(ALURegA.data.FLOAT16) >>
                                     imm); break;
                        } break;
                        case 0x80: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC64X32[destoff] = (ALURegA.
                                    data.UINT32) >> imm; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC32X64[destoff] = (ALURegA.data
                                    .INT32) >> imm; break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC32X64[destoff] = (float32_t)
                                    ((int64_t)(ALURegA.data.FLOAT32) >>
                                    imm); break;
                        } break;
                        case 0xC0: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC64X32[destoff] = (ALURegA.
                                    data.UINT64) >> imm; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC64X32[destoff] = (ALURegA.data
                                    .INT64) >> imm; break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC64X32[destoff] = (float64_t)
                                    ((int64_t)(ALURegA.data.FLOAT64) >>
                                    imm); break;
                        } break;
                }
        }
        else
        {return 1;}
}


//FIXME: C's right shifts are type sensitive and implementaiton defined
    ...
int doSRAs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if ((typeconverts(&ALURegA, dst , src1, off1) == 0) && (
            typeconverts(&ALURegB, dst , src2, off2) == 0))
        {
                switch(dsize)
                {
                        case 0x00: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
```

```
                                UINTVEC8X256[destoff] = (ALURegA.
                                data.UINT8) >> imm; break;
                            case 0x20: DLARFile[dst].data.
                                INTVEC8X256[destoff] = (ALURegA.data
                                .INT8) >> imm; break;
                            case 0x30: DLARFile[dst].data.
                                FLOATVEC8X256[destoff] = (float)((
                                int64_t)(ALURegA.data.FLOAT8) >> imm
                                ); break;
                    } break;
                    case 0x40: switch(dtype)
                    {
                            case 0x10: DLARFile[dst].data.
                                UINTVEC16X128[destoff] = (ALURegA.
                                data.UINT16) >> imm; break;
                            case 0x20: DLARFile[dst].data.
                                INTVEC16X128[destoff] = (ALURegA.
                                data.INT16) >> imm; break;
                            case 0x30: DLARFile[dst].data.
                                FLOATVEC16X128[destoff] = (float)((
                                int64_t)(ALURegA.data.FLOAT16) >>
                                imm); break;
                    } break;
                    case 0x80: switch(dtype)
                    {
                            case 0x10: DLARFile[dst].data.
                                UINTVEC64X32[destoff] = (ALURegA.
                                data.UINT32) >> imm; break;
                            case 0x20: DLARFile[dst].data.
                                INTVEC32X64[destoff] = (ALURegA.data
                                .INT32) >> imm; break;
                            case 0x30: DLARFile[dst].data.
                                FLOATVEC32X64[destoff] = (float)((
                                int64_t)(ALURegA.data.FLOAT32) >>
                                imm); break;
                    } break;
                    case 0xC0: switch(dtype)
                    {
                            case 0x10: DLARFile[dst].data.
                                UINTVEC64X32[destoff] = (ALURegA.
                                data.UINT64) >> imm; break;
                            case 0x20: DLARFile[dst].data.
                                INTVEC64X32[destoff] = (ALURegA.data
                                .INT64) >> imm; break;
                            case 0x30: DLARFile[dst].data.
                                FLOATVEC64X32[destoff] = (double)((
                                int64_t)(ALURegA.data.FLOAT64) >>
                                imm); break;
                    } break;
            }
        }
        else
        {return 1;}
}
```

```
int doSLTs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if ((typeconverts(&ALURegA, dst , src1, off1) == 0) && (
            typeconverts(&ALURegB, dst , src2, off2) == 0))
        {
                switch(dsize)
                {
                        case 0x00: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC8X256[destoff] = (ALURegA.
                                    data.UINT8) < (ALURegB.data.UINT8);
                                    break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC8X256[destoff] = (ALURegA.data
                                    .INT8) < (ALURegB.data.INT8);break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC8X256[destoff] = (ALURegA.
                                    data.FLOAT8) < (ALURegB.data.FLOAT8)
                                    ; break;

                        } break;
                        case 0x40: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC16X128[destoff] = (ALURegA.
                                    data.UINT16) < (ALURegB.data.UINT16)
                                    ; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC16X128[destoff] = (ALURegA.
                                    data.INT16) < (ALURegB.data.INT16);
                                    break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC16X128[destoff] = (ALURegA.
                                    data.FLOAT16) < (ALURegB.data.
                                    FLOAT16); break;

                        } break;
                        case 0x80: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC64X32[destoff] = (ALURegA.
                                    data.UINT32) < (ALURegB.data.UINT32)
                                    ; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC32X64[destoff] = (ALURegA.data
                                    .INT32) < (ALURegB.data.INT32);
                                    break;
```

```
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC32X64[destoff] = (ALURegA.
                                    data.FLOAT32) < (ALURegB.data.
                                    FLOAT32); break;
                        } break;
                        case 0xC0: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC64X32[destoff] = (ALURegA.
                                    data.UINT64) < (ALURegB.data.UINT64)
                                    ; break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC64X32[destoff] = (ALURegA.data
                                    .INT64) < (ALURegB.data.INT64);
                                    break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC64X32[destoff] = (ALURegA.
                                    data.FLOAT64) < (ALURegB.data.
                                    FLOAT64); break;
                        } break;

                }
        }
        else { return 1;}
}


//Arithmetic Instructions: Vector Mode
VALUOP(ADD, +, dst, src1, src2, off1, off2, destoff, imm)

VALUOP(SUB, -, dst, src1, src2, off1, off2, destoff, imm)

VALUOP(MUL, *, dst, src1, src2, off1, off2, destoff, imm)

VALUOP(DIV, /, dst, src1, src2, off1, off2, destoff, imm)

VALUBINOP(MOD, %, dst, src1, src2, off1, off2, destoff, imm)

VALUBINOP(AND, &, dst, src1, src2, off1, off2, destoff, imm)

VALUBINOP(OR, |, dst, src1, src2, off1, off2, destoff, imm)

VALUBINOP(XOR, ^, dst, src1, src2, off1, off2, destoff, imm)

int doNOTv(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{
        int i;
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if(typeconvertv(&ALULARA, dst , src1, off1) == 0)
        {
                for(i=0;i<LARWIDTH/dsize;i++)
```

```
{
        switch(dsize)
        {
        case 0x00: switch(dtype)
        {
                case 0x10: DLARFile[dst].data.
                    UINTVEC8X256[i] = ~ALULARA.data.
                    UINTVEC8X256[i]; break;
                case 0x20: DLARFile[dst].data.
                    INTVEC8X256[i] = ~ALULARA.data.
                    INTVEC8X256[i]; break;
                case 0x30: DLARFile[dst].data.
                    FLOATVEC8X256[i] = (float8_t)(~(
                    int64_t)ALULARA.data.FLOATVEC8X256[i
                    ]); break;
        } break;
        case 0x40: switch(dtype)
        {
                case 0x10: DLARFile[dst].data.
                    UINTVEC16X128[i] = ~ALULARA.data.
                    UINTVEC16X128[i]; break;
                case 0x20: DLARFile[dst].data.
                    INTVEC16X128[i] = ~ALULARA.data.
                    INTVEC16X128[i]; break;
                case 0x30: DLARFile[dst].data.
                    FLOATVEC16X128[i] = (float16_t)(~(
                    int64_t)ALULARA.data.FLOATVEC16X128[
                    i]); break;
        } break;
        case 0x80: switch(dtype)
        {
                case 0x10: DLARFile[dst].data.
                    UINTVEC32X64[i] = ~ALULARA.data.
                    UINTVEC32X64[i]; break;
                case 0x20: DLARFile[dst].data.
                    INTVEC32X64[i] = ~ALULARA.data.
                    INTVEC32X64[i]; break;
                case 0x30: DLARFile[dst].data.
                    FLOATVEC32X64[i] = (float32_t)(~(
                    int64_t)ALULARA.data.FLOATVEC32X64[i
                    ]); break;
        } break;
        case 0xC0: switch(dtype)
        {
                case 0x10: DLARFile[dst].data.
                    UINTVEC64X32[i] = ~ALULARA.data.
                    UINTVEC64X32[i]; break;
                case 0x20: DLARFile[dst].data.
                    INTVEC64X32[i] = ~ALULARA.data.
                    INTVEC64X32[i]; break;
                case 0x30: DLARFile[dst].data.
                    FLOATVEC64X32[i] = (float64_t)(~(
                    int64_t)ALULARA.data.FLOATVEC64X32[i
                    ]); break;
```

```
                                } break;
                                }
                        }
                        return 0;
                }
                else
                { return 1;}
}

//Vile extra casts to fix C's context-sensitive shift operators
int doSLLv(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{       int i;
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if((typeconvertv(&ALULARA, dst , src1, off1) == 0) && (
            typeconvertv(&ALULARB, dst , src2, off2) == 0))
        {
                for(i=0;i<LARWIDTH/dsize;i++)
                {
                        switch(dsize)
                        {
                                case 0x00: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                            UINTVEC8X256[i] = ALULARA.
                                            data.UINTVEC8X256[i] << imm;
                                             break;
                                        case 0x20: DLARFile[dst].data.
                                            INTVEC8X256[i] = (int8_t)((
                                            uint8_t)ALULARA.data.
                                            INTVEC8X256[i] << imm);
                                            break;
                                        case 0x30: DLARFile[dst].data.
                                            FLOATVEC8X256[i] = (float8_t
                                            )((uint64_t)ALULARA.data.
                                            FLOATVEC8X256[i] << imm);
                                            break;

                                } break;
                                case 0x40: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                            UINTVEC16X128[i] = ALULARA.
                                            data.UINTVEC16X128[i] << imm
                                            ; break;
                                        case 0x20: DLARFile[dst].data.
                                            INTVEC16X128[i] = (int16_t)
                                            ((uint16_t)ALULARA.data.
                                            INTVEC16X128[i] << imm);
                                            break;
                                        case 0x30: DLARFile[dst].data.
                                            FLOATVEC16X128[i] = (
```

```
                                                float16_t)((uint64_t)ALULARA
                                                .data.FLOATVEC16X128[i] <<
                                                imm); break;
                                    } break;
                                    case 0x80: switch(dtype)
                                    {
                                            case 0x10: DLARFile[dst].data.
                                                UINTVEC32X64[i] = ALULARA.
                                                data.UINTVEC32X64[i] << imm;
                                                 break;
                                            case 0x20: DLARFile[dst].data.
                                                INTVEC32X64[i] = (int32_t)((
                                                uint32_t)ALULARA.data.
                                                INTVEC32X64[i] << imm);
                                                break;
                                            case 0x30: DLARFile[dst].data.
                                                FLOATVEC32X64[i] = (
                                                float32_t)((uint64_t)ALULARA
                                                .data.FLOATVEC32X64[i] <<
                                                imm); break;
                                    } break;
                                    case 0xC0: switch(dtype)
                                    {
                                            case 0x10: DLARFile[dst].data.
                                                UINTVEC64X32[i] = ALULARA.
                                                data.UINTVEC64X32[i] << imm;
                                                 break;
                                            case 0x20: DLARFile[dst].data.
                                                INTVEC64X32[i] = (int64_t)((
                                                uint64_t)ALULARA.data.
                                                INTVEC64X32[i] << imm);
                                                break;
                                            case 0x30: DLARFile[dst].data.
                                                FLOATVEC64X32[i] = (
                                                float64_t)((uint64_t)ALULARA
                                                .data.FLOATVEC64X32[i] <<
                                                imm); break;
                                    } break;

                        }
                }
                return 0;
        }
        else
        {return 1;}
}


//Vile extra casts to fix C's context-sensitive shift operators
int doSRAv(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{       int i;
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
```

```
DLARFile[dst].flags &= 0x08;
if((typeconvertv(&ALULARA, dst , src1, off1) == 0) && (
    typeconvertv(&ALULARB, dst , src2, off2) == 0))
{
        for(i=0;i<LARWIDTH/dsize;i++)
        {
                switch(dsize)
                {
                        case 0x00: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC8X256[i] = (uint8_t)
                                    ((int8_t)ALULARA.data.
                                    UINTVEC8X256[i] >> imm);
                                    break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC8X256[i] = ALULARA.
                                    data.INTVEC8X256[i] >> imm;
                                    break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC8X256[i] = (float8_t
                                    )((int64_t)ALULARA.data.
                                    FLOATVEC8X256[i] >> imm);
                                    break;

                        } break;
                        case 0x40: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC16X128[i] = (uint16_t
                                    )((int16_t)ALULARA.data.
                                    UINTVEC16X128[i] >> imm);
                                    break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC16X128[i] = ALULARA.
                                    data.INTVEC16X128[i] >> imm;
                                     break;
                                case 0x30: DLARFile[dst].data.
                                    FLOATVEC16X128[i] = (
                                    float16_t)((int64_t)ALULARA.
                                    data.FLOATVEC16X128[i] >>
                                    imm); break;
                        } break;
                        case 0x80: switch(dtype)
                        {
                                case 0x10: DLARFile[dst].data.
                                    UINTVEC32X64[i] = (uint32_t)
                                    ((int8_t)ALULARA.data.
                                    UINTVEC32X64[i] >> imm);
                                    break;
                                case 0x20: DLARFile[dst].data.
                                    INTVEC32X64[i] = ALULARA.
                                    data.INTVEC32X64[i] >> imm;
                                    break;
```

```
                                        case 0x30: DLARFile[dst].data.
                                           FLOATVEC32X64[i] = (
                                           float32_t)((int64_t)ALULARA.
                                           data.FLOATVEC32X64[i] >> imm
                                           ); break;
                                } break;
                                case 0xC0: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                           UINTVEC64X32[i] = (uint64_t)
                                           ((int64_t)ALULARA.data.
                                           UINTVEC64X32[i] >> imm);
                                           break;
                                        case 0x20: DLARFile[dst].data.
                                           INTVEC64X32[i] = ALULARA.
                                           data.INTVEC64X32[i] >> imm;
                                           break;
                                        case 0x30: DLARFile[dst].data.
                                           FLOATVEC64X32[i] = (
                                           float64_t)((int64_t)ALULARA.
                                           data.FLOATVEC64X32[i] >> imm
                                           ); break;
                                } break;

                        }
                }
                return 0;
        }
        else
        {return 1;}
}
//Vile extra casts to fix C's context-sensitive shift operators
int doSRLv(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, uint8_t imm)
{       int i;
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if((typeconvertv(&ALULARA, dst , src1, off1) == 0) && (
           typeconvertv(&ALULARB, dst , src2, off2) == 0))
        {
                for(i=0;i<LARWIDTH/dsize;i++)
                {
                        switch(dsize)
                        {
                                case 0x00: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                           UINTVEC8X256[i] = ALULARA.
                                           data.UINTVEC8X256[i] >> imm;
                                            break;
                                        case 0x20: DLARFile[dst].data.
                                           INTVEC8X256[i] = (int8_t)((
                                           uint8_t)ALULARA.data.
```

```
            INTVEC8X256[i] >> imm);
            break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC8X256[i] = (float8_t
            )((uint64_t)ALULARA.data.
            FLOATVEC8X256[i] >> imm);
            break;

} break;
case 0x40: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC16X128[i] = ALULARA.
            data.UINTVEC16X128[i] >> imm
            ; break;
        case 0x20: DLARFile[dst].data.
            INTVEC16X128[i] = (int16_t)
            ((uint16_t)ALULARA.data.
            INTVEC16X128[i] >> imm);
            break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC16X128[i] = (
            float16_t)((uint64_t)ALULARA
            .data.FLOATVEC16X128[i] >>
            imm); break;
} break;
case 0x80: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC32X64[i] = ALULARA.
            data.UINTVEC32X64[i] >> imm;
             break;
        case 0x20: DLARFile[dst].data.
            INTVEC32X64[i] = (int32_t)((
            uint32_t)ALULARA.data.
            INTVEC32X64[i] >> imm);
            break;
        case 0x30: DLARFile[dst].data.
            FLOATVEC32X64[i] = (
            float32_t)((uint64_t)ALULARA
            .data.FLOATVEC32X64[i] >>
            imm); break;
} break;
case 0xC0: switch(dtype)
{
        case 0x10: DLARFile[dst].data.
            UINTVEC64X32[i] = ALULARA.
            data.UINTVEC64X32[i] >> imm;
             break;
        case 0x20: DLARFile[dst].data.
            INTVEC64X32[i] = (int64_t)((
            uint64_t)ALULARA.data.
            INTVEC64X32[i] >> imm);
            break;
```

```
                                        case 0x30: DLARFile[dst].data.
                                           FLOATVEC64X32[i] = (
                                           float64_t)((uint64_t)ALULARA
                                           .data.FLOATVEC64X32[i] >>
                                           imm); break;
                                } break;

                        }
                }
        }
        else
        {return 1;}
}



//I'm not even sure if SLT is fully defined for vectors...
//      Currently makes a full vector of results... sum for scalar.
int doSLTv(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, uint8_t imm)
{
        int i;
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & 0xC0)+3));
        uint8_t dtype = DLARFile[dst].flags&0x30;
        DLARFile[dst].flags &= 0x08;
        if ((typeconvertv(&ALULARA, dst , src1, off1) == 0) && (
            typeconvertv(&ALULARB, dst , src2, off2) == 0))
        {
                for(i=0;i<2048/dsize;i++)
                {
                        switch(dsize)
                        {
                                case 0x00: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                           UINTVEC8X256[i] = ALULARA.
                                           data.UINTVEC8X256[i] <
                                           ALULARB.data.UINTVEC8X256[i
                                           ]; break;
                                        case 0x20: DLARFile[dst].data.
                                           INTVEC8X256[i] = ALULARA.
                                           data.INTVEC8X256[i] <
                                           ALULARB.data.INTVEC8X256[i];
                                            break;
                                        case 0x30: DLARFile[dst].data.
                                           FLOATVEC8X256[i] = ALULARA.
                                           data.FLOATVEC8X256[i] <
                                           ALULARB.data.FLOATVEC8X256[i
                                           ]; break;
                                } break;
                                case 0x40: switch(dtype)
                                {
                                        case 0x10: DLARFile[dst].data.
                                           UINTVEC16X128[i] = ALULARA.
```

```
                        data.UINTVEC16X128[i] <
                        ALULARB.data.UINTVEC16X128[i
                        ]; break;
                    case 0x20: DLARFile[dst].data.
                        INTVEC16X128[i] = ALULARA.
                        data.INTVEC16X128[i] <
                        ALULARB.data.INTVEC16X128[i
                        ]; break;
                    case 0x30: DLARFile[dst].data.
                        FLOATVEC16X128[i] = ALULARA.
                        data.FLOATVEC16X128[i] <
                        ALULARB.data.FLOATVEC16X128[
                        i]; break;
                } break;
                case 0x80: switch(dtype)
                {
                    case 0x10: DLARFile[dst].data.
                        UINTVEC32X64[i] = ALULARA.
                        data.UINTVEC32X64[i] <
                        ALULARB.data.UINTVEC32X64[i
                        ]; break;
                    case 0x20: DLARFile[dst].data.
                        INTVEC32X64[i] = ALULARA.
                        data.INTVEC32X64[i] <
                        ALULARB.data.INTVEC32X64[i];
                         break;
                    case 0x30: DLARFile[dst].data.
                        FLOATVEC32X64[i] = ALULARA.
                        data.FLOATVEC32X64[i] <
                        ALULARB.data.FLOATVEC32X64[i
                        ]; break;
                } break;
                case 0xC0: switch(dtype)
                {
                    case 0x10: DLARFile[dst].data.
                        UINTVEC64X32[i] = ALULARA.
                        data.UINTVEC64X32[i] <
                        ALULARB.data.UINTVEC64X32[i
                        ]; break;
                    case 0x20: DLARFile[dst].data.
                        INTVEC64X32[i] = ALULARA.
                        data.INTVEC64X32[i] <
                        ALULARB.data.INTVEC64X32[i];
                         break;
                    case 0x30: DLARFile[dst].data.
                        FLOATVEC64X32[i] = ALULARA.
                        data.FLOATVEC64X32[i] <
                        ALULARB.data.FLOATVEC64X32[i
                        ]; break;
                } break;
            }
        }
    }
    else
```

```
        { return 1;}
}



//Loads
LDOP(8U, 0x10, dst, src1, src2, off1, off2, destoff, imm)
LDOP(16U, 0x50, dst, src1, src2, off1, off2, destoff, imm)
LDOP(32U, 0x90, dst, src1, src2, off1, off2, destoff, imm)
LDOP(64U, 0xD0, dst, src1, src2, off1, off2, destoff, imm)

LDOP(8I, 0x20, dst, src1, src2, off1, off2, destoff, imm)
LDOP(16I, 0x60, dst, src1, src2, off1, off2, destoff, imm)
LDOP(32I, 0xC0, dst, src1, src2, off1, off2, destoff, imm)
LDOP(64I, 0xE0, dst, src1, src2, off1, off2, destoff, imm)

LDOP(8F, 0x30, dst, src1, src2, off1, off2, destoff, imm)
LDOP(16F, 0x70, dst, src1, src2, off1, off2, destoff, imm)
LDOP(32F, 0xB0, dst, src1, src2, off1, off2, destoff, imm)
LDOP(64F, 0XF0, dst, src1, src2, off1, off2, destoff, imm)


//Stores
STOP(8U, 0x10, dst, src1, src2, off1, off2, destoff, imm)
STOP(16U, 0x50, dst, src1, src2, off1, off2, destoff, imm)
STOP(32U, 0x90, dst, src1, src2, off1, off2, destoff, imm)
STOP(64U, 0xD0, dst, src1, src2, off1, off2, destoff, imm)

STOP(8I, 0x20, dst, src1, src2, off1, off2, destoff, imm)
STOP(16I, 0x60, dst, src1, src2, off1, off2, destoff, imm)
STOP(32I, 0xC0, dst, src1, src2, off1, off2, destoff, imm)
STOP(64I, 0xE0, dst, src1, src2, off1, off2, destoff, imm)

STOP(8F, 0x30, dst, src1, src2, off1, off2, destoff, imm)
STOP(16F, 0x70, dst, src1, src2, off1, off2, destoff, imm)
STOP(32F, 0xB0, dst, src1, src2, off1, off2, destoff, imm)
STOP(64F, 0XF0, dst, src1, src2, off1, off2, destoff, imm)




//Not 100% Sure this is working as intended
        // This is another place where the macro system could have been
            more elegant, but replication to the rescue...
int doSEL(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1, uint8_t
     off2, uint8_t destoff, uint8_t imm)
{
        int testcond;
        int srctyp= DLARFile[dst].flags & TYPMASK;
        switch(DLARFile[dst].flags & WDSZMASK)
        {
                case SZ8:
                        switch(srctyp)
                        {
```

```
        case TUINT:
                testcond = (int)DLARFile[dst].data.
                   UINTVEC8X256[destoff]; break;
        case TINT:
                testcond = (int)DLARFile[dst].data.
                   INTVEC8X256[destoff]; break;
        case TFLOAT:
                testcond = (int)DLARFile[dst].data.
                   FLOATVEC8X256[destoff]; break;
        }
break;
case SZ16:
        switch(srctyp)
        {
                case TUINT:
                        testcond = (int)DLARFile[dst].
                           data.UINTVEC16X128[destoff];
                            break;
                case TINT:
                        testcond = (int)DLARFile[dst].
                           data.INTVEC16X128[destoff];
                           break;
                case TFLOAT:
                        testcond = (int)DLARFile[dst].
                           data.FLOATVEC16X128[destoff
                           ]; break;
        }
break;
case SZ32:
        switch(srctyp)
        {
                case TUINT:
                        testcond = (int)DLARFile[dst].
                           data.UINTVEC32X64[destoff];
                           break;
                case TINT:
                        testcond = (int)DLARFile[dst].
                           data.INTVEC32X64[destoff];
                           break;
                case TFLOAT:
                        testcond = (int)DLARFile[dst].
                           data.FLOATVEC32X64[destoff];
                            break;
        }
break;
case SZ64:
        switch(srctyp)
        {
                case TUINT:
                        testcond = (int)DLARFile[dst].
                           data.UINTVEC64X32[destoff];
                           break;
                case TINT:
                        testcond = (int)DLARFile[dst].
```

```
                                                data.INTVEC64X32[destoff];
                                                break;
                                case TFLOAT:
                                        testcond = (int)DLARFile[dst].
                                                data.FLOATVEC64X32[destoff];
                                                 break;
                        }
                break;
                }
        if(testcond != 0)//Set PC to first target.. just being
            redundantly explicit
        {
                DLARFile[1].data.UINTVEC64X32[0]=ILARFile[src1].instr[
                    off1];
        }
        else//Set PC to second target
        {
                DLARFile[1].data.UINTVEC64X32[0]=ILARFile[src2].instr[
                    off2];
        }
}

//Fetch an ILAR-width line of instructions from memory
        //No compression currently implimented, just does a linear load.
int doFETCH(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
    uint8_t off2, uint8_t destoff, uint8_t imm)
{
        int i;
        int address;
        ADDRCALC(dst, src1, src2, off1, off2, destoff, imm)
        ILARFile[dst].addr=address;
        for(i=0;i<255;i++)
        {
                ILARFile[dst].instr[i]=RAMFile[address+i];
        }
}
```

Listing A.2: LARKemNoSIMD.h

```
//The "NoSIMD" version
// There is a small chance that doing -mfpmath flags appropriately would
     do well with this...
#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <strings.h>
#include "aluops.h"
#include "memops.h"


//Masks for the status bits
#define WDSZMASK 0xC0
#define TYPMASK  0x30
#define DIRTYMASK 0x08
```

```
//Constants
#define LARWIDTH 2048
#define MEMSZ 4096

#define TUINT 0x10
#define TINT 0x20
#define TFLOAT 0x30

#define SZ8 0x00
#define SZ16 0x40
#define SZ32 0x80
#define SZ64 0xC0

#define ASMBUFSZ 255


//Normalize nomenclature, make it easy to swap other representations in
   for unusual width floats
// But everything is really a machine-native float, because screw adding
    another layer to make baby-floats
#define float8_t float
#define float16_t float
#define float32_t float
#define float64_t double


//This is a DLAR. Fear it.
typedef struct
{
        uint8_t flags;
        uint64_t addr;
        //All the same size and shape...
        union
        {
                uint8_t UINTVEC8X256[256];
                uint16_t UINTVEC16X128[128];
                uint32_t UINTVEC32X64[64];
                uint64_t UINTVEC64X32[32];

                int8_t INTVEC8X256[256];
                int16_t INTVEC16X128[128];
                int32_t INTVEC32X64[64];
                int64_t INTVEC64X32[32];

                //Are <32 bit floats going to have to be assembled by
                   hand?
                //FIXME for now blowing my alignment and reusing floats
                float8_t FLOATVEC8X256[256];
                float16_t FLOATVEC16X128[128];
                float32_t FLOATVEC32X64[64];
                float64_t FLOATVEC64X32[32];

        }data;
```

```
}DLAR_t;

//A scalar buffer with type tags to use for the scalar ALU path
typedef struct
{
        uint8_t flags;
        union
        {
                uint8_t UINT8;
                uint16_t UINT16;
                uint32_t UINT32;
                uint64_t UINT64;

                int8_t INT8;
                int16_t INT16;
                int32_t INT32;
                int64_t INT64;

                //Are <32 bit floats going to have to be assembled by
                   hand?
                //FIXME for now, three names for the same "float"
                float8_t FLOAT8;
                float16_t FLOAT16;
                float32_t FLOAT32;
                float64_t FLOAT64;
        }data;
}scalarbuf_t;

//ILAR Structure ILAR
typedef struct
{
        uint64_t addr;
        uint64_t instr[2048];
}ILAR_t;

//Prototype for the horrible lar-width conversion function
int typeconvertv(DLAR_t *buf, int dst , int src, uint8_t srcoff);
int typeconverts(scalarbuf_t *buf, int dst , int src, uint8_t srcoff);


/*

doLOAD8U
doLOAD16U
doLOAD32U
doLOAD64U
doLOAD8I
doLOAD16I
doLOAD32I
doLOAD64I
doLOAD8F
doLOAD16F
doLOAD32F
doLOAD64F
```

```
doSTORE8U
doSTORE16U
doSTORE32U
doSTORE64U
doSTORE8I
doSTORE16I
doSTORE32I
doSTORE64I
doSTORE8F
doSTORE16F
doSTORE32F
doSTORE64F


void doADDs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, int imm);
void doSUBs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, int imm);
void doMULs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, int imm);
void doDIVs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, int imm);
void doMODs(uint8_t dst, uint8_t src1, uint8_t src2, uint8_t off1,
   uint8_t off2, uint8_t destoff, int imm);

doANDs
doORs
doXORs
doNEGs
doSLLs
doSRAs
doSRLs

doSLTs

doADDv
doSUBv
doMULv
doDIVv
doMODv

doANDv
doORv
doXORv
doNEGv
doSLLv
doSRAv
doSRLv

doSLTv


doSEL
```

```
//doCALL
//doRETURN
*/
```

Listing A.3: DLARConvert.h

```
#include "LARKemNoSIMD.h"
#include <math.h>

//Ia Ia accesessormacrothulu fhtagn!
// This loads an item in an ALU DLAR buffer (buf) with typeconverted data from SRC
// DSTTYP is the destination type (DLAR type, not C type)
// DSTOFF is an offset in the destination (integer)
// CTYP is the C type to be converted to
// SRC is a location in the DLARFile (integer)
// SRCOFF is an offset in the source (integer)
#define DLARCONVERTV(DESTTYP, DESTOFF, CTYP, SRC, SRCOFF)\
switch(srcsize)\
{\
	case SZ8: \
		switch(srctyp)\
		{\
		case TUINT:\
			buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].data.
			    UINTVEC8X256[(SRCOFF)]; break;\
		case TINT:\
			buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].data.
			    INTVEC8X256[(SRCOFF)]; break;\
		case TFLOAT:\
			buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].data.
			    FLOATVEC8X256[(SRCOFF)]; break;\
		}\
	break;\
	case SZ16:\
		switch(srctyp)\
		{\
			case TUINT:\
				buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
				    data.UINTVEC16X128[(SRCOFF)]; break;\
			case TINT:\
				buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
				    data.INTVEC16X128[(SRCOFF)]; break;\
			case TFLOAT:\
				buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
				    data.FLOATVEC16X128[(SRCOFF)]; break;\
		}\
	break;\
	case SZ32: \
		switch(srctyp)\
		{\
			case TUINT:\
				buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
				    data.UINTVEC32X64[(SRCOFF)]; break;\
			case TINT:\
```

```
                    buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
                        data.INTVEC32X64[(SRCOFF)]; break;\
                case TFLOAT:\
                    buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
                        data.FLOATVEC32X64[(SRCOFF)]; break;\
            }\
        break;\
        case SZ64: \
            switch(srctyp)\
            {\
                case TUINT:\
                    buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
                        data.UINTVEC64X32[(SRCOFF)]; break;\
                case TINT:\
                    buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
                        data.INTVEC64X32[(SRCOFF)]; break;\
                case TFLOAT:\
                    buf->data.DESTTYP[(DESTOFF)] = (CTYP) DLARFile[(SRC)].
                        data.FLOATVEC64X32[(SRCOFF)]; break;\
            }\
        break;\
}\

//A silly little macro to simplify filling the remainder of a DLAR on a
    downconvert
#define DLARFILLV(DESTTYP, FILL)\
while(i<LARWIDTH/SRCSZ)\
{\
        buf->data.DESTTYP[i] = FILL;\
        i++;\
}\


//Differs from vector version only by having one fewer levels of indirection on
    the buffer
#define DLARCONVERTS(DESTTYP, CTYP, SRC, SRCOFF)\
switch(srcsize)\
{\
        case SZ8: \
            switch(srctyp)\
            {\
            case TUINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.UINTVEC8X256[(
                    SRCOFF)]; break;\
            case TINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.INTVEC8X256[(
                    SRCOFF)]; break;\
            case TFLOAT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.FLOATVEC8X256[(
                    SRCOFF)]; break;\
            }\
        break;\
        case SZ16:\
            switch(srctyp)\
```

```
        {\
            case TUINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.
                    UINTVEC16X128[(SRCOFF)]; break;\
            case TINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.INTVEC16X128
                    [(SRCOFF)]; break;\
            case TFLOAT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.
                    FLOATVEC16X128[(SRCOFF)]; break;\
        }\
    break;\
    case SZ32: \
        switch(srctyp)\
        {\
            case TUINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.UINTVEC32X64
                    [(SRCOFF)]; break;\
            case TINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.INTVEC32X64
                    [(SRCOFF)]; break;\
            case TFLOAT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.
                    FLOATVEC32X64[(SRCOFF)]; break;\
        }\
    break;\
    case SZ64: \
        switch(srctyp)\
        {\
            case TUINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.UINTVEC64X32
                    [(SRCOFF)]; break;\
            case TINT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.INTVEC64X32
                    [(SRCOFF)]; break;\
            case TFLOAT:\
                buf->DESTTYP = (CTYP) DLARFile[(SRC)].data.
                    FLOATVEC64X32[(SRCOFF)]; break;\
        }\
    break;\
}\
```

Listing A.4: converter.c

```
#include "DLARConvert.h"

//FIXME: The scalar and vector versions really should be derived from
    the same pool of functions and macros...

//This is the full-vector typeconverter
// It converts a DLARFile entry "src" to the type of a DLARFile entry "
    dst", and puts it in "buf"
// The DLARCONVERT macro is another couple layers of switches and evil,
    abstracted out
```

```
// Returns 0 on success, anything else is a fail. (1 for invalid size, 2
    for invalid type, 3 for alignment problem)
int typeconvertv(DLAR_t *buf, int dst , int src, uint8_t srcoff)
{
        //reusable iterator
        int i;

        //Convert WDSZ fields to actual wordsizes in bits
        int dstsize = pow(2, ((DLARFile[dst].flags & WDSZMASK)+3));
        int srcsize = pow(2, ((DLARFile[src].flags & WDSZMASK)+3));
        uint8_t dsttyp = DLARFile[dst].flags&TYPMASK;
        uint8_t srctyp = DLARFile[src].flags&TYPMASK;


        //Set ALU buffer types to match destination DLAR
        buf->flags = DLARFile[dst].flags;

    if(dstsize > srcsize)//Upconvert
    {
    if(((LARWIDTH/srcsize)-srcoff) >= (LARWIDTH/dstsize))
    {
    switch(dstsize)
    {
    case 8:
    switch(dsttyp)
    {
    case TUINT:
        for(i=0;i<LARWIDTH/8;i++)
        {
            DLARCONVERTV(UINTVEC8X256, i, uint8_t, src, i+srcoff)
        }
    break;
    case TINT:
        for(i=0;i<LARWIDTH/8;i++)
        {
            DLARCONVERTV(INTVEC8X256, i, int8_t, src, i+srcoff)
        }
    break;
    case TFLOAT:
        for(i=0;i<LARWIDTH/8;i++)
        {
            DLARCONVERTV(FLOATVEC8X256, i, float8_t, src, i+srcoff)
        }
    break;
    default:
        return 2;
    break;
    }
    break;
    case 16:
    switch(dsttyp)
    {
    case TUINT:
        for(i=0;i<LARWIDTH/16;i++)
```

```
    {
        DLARCONVERTV(UINTVEC16X128, i, uint16_t, src, i+srcoff)
    }
break;
case TINT:
    for(i=0;i<LARWIDTH/16;i++)
    {
        DLARCONVERTV(INTVEC16X128, i, int16_t, src, i+srcoff)
    }
break;
case TFLOAT:
    for(i=0;i<LARWIDTH/16;i++)
    {
        DLARCONVERTV(FLOATVEC16X128, i, float16_t, src, i+srcoff)
    }
break;
default: return 2;
break;
}
break;
case 32:
switch(dsttyp)
{
case TUINT:
    for(i=0;i<LARWIDTH/32;i++)
    {
        DLARCONVERTV(UINTVEC32X64, i, uint32_t, src, i+srcoff)
    }
break;
case TINT:
    for(i=0;i<LARWIDTH/32;i++)
    {
        DLARCONVERTV(INTVEC32X64, i, int32_t, src, i+srcoff)
    }
break;
case TFLOAT:
    for(i=0;i<LARWIDTH/32;i++)
    {
        DLARCONVERTV(FLOATVEC32X64, i, float32_t, src, i+srcoff)
    }
break;
default: return 2;
break;
}
break;
case 64:
switch(dsttyp)
{
case TUINT:
    for(i=0;i<LARWIDTH/64;i++)
    {
        DLARCONVERTV(UINTVEC64X32, i, uint64_t, src, i+srcoff)
    }
break;
```

```
case TINT:
    for(i=0;i<LARWIDTH/64;i++)
    {
        DLARCONVERTV(INTVEC64X32, i, int64_t, src, i+srcoff)
    }
break;
case TFLOAT:
    for(i=0;i<LARWIDTH/64;i++)
    {
        DLARCONVERTV(FLOATVEC64X32, i, float64_t, src, i+srcoff)
    }
break;
default:
    return 2;
break;
}
break;
default: return 1;
}
}
else
{
return 3; //Not enough elements in SRC
}
}
else //Downconvert
//These cases do "As many as are available, starting from [0], then
   pad out with 0"
{
switch(dstsize)
{
case 8:
switch(dsttyp)
{
case TUINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(UINTVEC8X256, i, uint8_t, src, i)
    DLARFILL(UINTVEC8X256, 0)
}
break;
case TINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(INTVEC8X256, i, int8_t, src, i)
    DLARFILL(INTVEC8X256, 0)
}
break;
case TFLOAT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(FLOATVEC8X256, i, float8_t, src, i)
    DLARFILL(FLOATVEC8X256, 0.0)
}
```

```
break;
default:
return 2;
break;
}


break;
case 16:
switch(dsttyp)
{
case TUINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(UINTVEC16X128, i, uint16_t, src, i)
    DLARFILL(UINTVEC16X128, 0)
}
break;
case TINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(INTVEC16X128, i, int16_t, src, i)
    DLARFILL(INTVEC16X128, 0)
}
break;
case TFLOAT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(FLOATVEC16X128, i, float16_t, src, i)
    DLARFILL(FLOATVEC16X128, 0.0)
}
break;
default: return 2;
break;
}
break;
case 32:
switch(dsttyp)
{
case TUINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(UINTVEC32X64, i, uint32_t, src, i)
    DLARFILL(UINTVEC32X64, 0)
}
break;
case TINT:
for(i=0;i<LARWIDTH/srcsize;i++)
{
    DLARCONVERTV(INTVEC32X64, i, int32_t, src, i)
    DLARFILL(INTVEC32X64, 0)
}
break;
case TFLOAT:
```

```
    for(i=0;i<LARWIDTH/srcsize;i++)
    {
        DLARCONVERTV(FLOATVEC32X64, i, int32_t, src, i)
        DLARFILL(FLOATVEC32X64, 0.0)
    }
    break;
    default: return 2;
    break;
    }
    break;
    case 64:
    switch(dsttyp)
    {
    case TUINT:
    for(i=0;i<LARWIDTH/srcsize;i++)
    {
        DLARCONVERTV(UINTVEC64X32, i, uint64_t, src, i)
        DLARFILL(UINTVEC64X32, 0)
    }
    break;
    case TINT:
    for(i=0;i<LARWIDTH/srcsize;i++)
    {
        DLARCONVERTV(INTVEC64X32, i, int64_t, src, i)
        DLARFILL(INTVEC64X32, 0)
    }
    break;
    case TFLOAT:
    for(i=0;i<LARWIDTH/srcsize;i++)
    {
        DLARCONVERTV(FLOATVEC64X32, i, float64_t, src, i)
        DLARFILL(FLOATVEC64X32, 0.0)
    }
    break;
    default:
    return 2;
    break;
    }
    break;
    default: return 1;
    }
    }
return 0;
}

// A scalar-only typeconvert function --
// The vector case is completely seprate because branching overhead is
   OBSCENE on this operation (and I wrote the vector version first)
// Also, this one gets to skip a lot of tests and suchlike
int typeconverts(scalarbuf_t *buf, int dst , int src, uint8_t srcoff)
{
        //Convert WDSZ fields to actual wordsizes in bits
        int dstsize = pow(2, ((DLARFile[dst].flags & WDSZMASK)+3));
        int srcsize = pow(2, ((DLARFile[src].flags & WDSZMASK)+3));
```

```
    uint8_t dsttyp = DLARFile[dst].flags&TYPMASK;
    uint8_t srctyp = DLARFile[src].flags&TYPMASK;


    //Set ALU buffer types to match destination DLAR
    buf->flags = DLARFile[dst].flags;

if(dstsize > srcsize)//Upconvert
{
switch(dstsize)
{
case 8:
switch(dsttyp)
{
case TUINT:
    DLARCONVERTS(UINT8, uint8_t, src, srcoff)
break;
case TINT:
    DLARCONVERTS(INT8, int8_t, src, srcoff)
break;
case TFLOAT:
    DLARCONVERTS(FLOAT8, float8_t, src, srcoff)
break;
default:
    return 2;
break;
}
break;
case 16:
switch(dsttyp)
{
case TUINT:
    DLARCONVERTS(UINT16, uint16_t, src, srcoff)
break;
case TINT:
    DLARCONVERTS(INT16, int16_t, src, srcoff)
break;
case TFLOAT:
    DLARCONVERTS(FLOAT16, float16_t, src, srcoff)

break;
default: return 2;
break;
}
break;
case 32:
switch(dsttyp)
{
case TUINT:
    DLARCONVERTS(UINT32, uint32_t, src, srcoff)

break;
case TINT:
    DLARCONVERTS(INT32, int32_t, src, srcoff)
```

```
break;
case TFLOAT:
    DLARCONVERTS(FLOAT32, float32_t, src, srcoff)
break;
default: return 2;
break;
}
break;
case 64:
switch(DLARFile[dst].flags&TYPMASK)
{
case TUINT:
    DLARCONVERTS(UINT64, uint64_t, src, srcoff)
break;
case TINT:
    DLARCONVERTS(INT64, int64_t, src, srcoff)
break;
case TFLOAT:
        DLARCONVERTS(FLOAT64, float64_t, src, srcoff)
break;
default:
    return 2;
break;
}
break;
default: return 1;
}
}
else //Downconvert
//These cases do "As many as are available, starting from [0], then
   pad out with 0"
{
switch(dstsize)
{
case 8:
switch(dsttyp)
{
case TUINT:
        DLARCONVERTS(UINT8, uint8_t, src, srcoff)
break;
case TINT:
        DLARCONVERTS(INT8, int8_t, src, srcoff)
break;
case TFLOAT:
        DLARCONVERTS(FLOAT8, float8_t, src, srcoff)
break;
default:
    return 2;
break;
}


break;
case 16:
```

```
switch(dsttyp)
{
case TUINT:
        DLARCONVERTS(UINT16, uint16_t, src, srcoff)
break;
case TINT:
        DLARCONVERTS(INT16, int16_t, src, srcoff)
break;
case TFLOAT:
        DLARCONVERTS(FLOAT16, float16_t, src, srcoff)
break;
default: return 2;
break;
}
break;
case 32:
switch(dsttyp)
{
case TUINT:
    DLARCONVERTS(UINT32, uint32_t, src, srcoff)
break;
case TINT:
    DLARCONVERTS(INT32, int32_t, src, srcoff)
break;
case TFLOAT:
    DLARCONVERTS(FLOAT32, int32_t, src, srcoff)
break;
default: return 2;
break;
}
break;
case 64:
switch(dsttyp)
{
case TUINT:
        DLARCONVERTS(UINT64, uint64_t, src, srcoff)
break;
case TINT:
        DLARCONVERTS(INT64, int64_t, src, srcoff)
break;
case TFLOAT:
        DLARCONVERTS(FLOAT64, float64_t, src, srcoff)
break;
default:
    return 2;
break;
}
break;
default: return 1;
break;
}
}

    return 0;
```

```
}
```

Listing A.5: aluops.h

```
// Simple two-operand operations are constructed from nested macros...
// Operations which do not fit the pattern of these nested macros are
   made using hand-modified versions of the expanded code

//Wherever operations are disallowed in C, casting to int64_t

//Macro'd prototype for simple scalar ALU operation
#define SALUOP(OPNAME, OP, DST, SRC1, SRC2, OFF1, OFF2, DSTOFF, IMM)\
int do##OPNAME##s(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DSTOFF, uint8_t IMM)\
{\
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & WDSZMASK)+3));\
        uint8_t dtype = DLARFile[DST].flags&TYPMASK;\
        DLARFile[DST].flags &= 0x08;\
        if ((typeconverts(&ALURegA, DST , SRC1, OFF1) == 0) && (
           typeconverts(&ALURegB, DST , SRC2, OFF2) == 0))\
        {\
                DLAROPTYPS(OFF1, OFF2, DST, DSTOFF, OP)\
                return 0;\
        }\
        else\
        {\
                return 1;\
        }\
}\

//Macro'd prototype for scalar ALU operations that C doesn't like doing
   on non-integer bit patterns
#define SALUBINOP(OPNAME, OP, DST, SRC1, SRC2, OFF1, OFF2, DSTOFF, IMM)\
int do##OPNAME##s(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DSTOFF, uint8_t IMM)\
{\
        uint8_t dsize = pow(2, ((DLARFile[DST].flags & 0xC0)+3));\
        uint8_t dtype = DLARFile[DST].flags&0x30; DLARFile[DST].flags &=
            0x08;\
        if ((typeconverts(&ALURegA, DST , SRC1, OFF1) == 0) && (
           typeconverts(&ALURegB, DST , SRC2, OFF2) == 0))\
        {\
                switch(dsize)\
                {\
                        case 0x00: switch(dtype)\
                        {\
                                case 0x10: DLARFile[DST].data.
                                   UINTVEC8X256[DSTOFF] = (ALURegA.data
                                   .UINT8) OP (ALURegB.data.UINT8);
                                   break;\
                                case 0x20: DLARFile[DST].data.
                                   INTVEC8X256[DSTOFF] = (ALURegA.data.
                                   INT8) OP (ALURegB.data.INT8); break
                                   ;\
```

```
        case 0x30: DLARFile[DST].data.
           FLOATVEC8X256[DSTOFF] = (float8_t)
           (((int64_t)ALURegA.data.FLOAT8) OP
           ((int64_t)ALURegB.data.FLOAT8));
           break;\
} break;\
case 0x40: switch(dtype)\
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC16X128[DSTOFF] = (ALURegA.
           data.UINT16) OP (ALURegB.data.UINT16
           ); break;\
        case 0x20: DLARFile[DST].data.
           INTVEC16X128[DSTOFF] = (ALURegA.data
           .INT16) OP (ALURegB.data.INT16);
           break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC16X128[DSTOFF] = (float16_t)
           (((int64_t)ALURegA.data.FLOAT16) OP
           ((int64_t)ALURegB.data.FLOAT16));
           break;\
} break;\
case 0x80: switch(dtype)\
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC64X32[DSTOFF] = (ALURegA.data
           .UINT32) OP (ALURegB.data.UINT32);
           break;\
        case 0x20: DLARFile[DST].data.
           INTVEC32X64[DSTOFF] = (ALURegA.data.
           INT32) OP (ALURegB.data.INT32);
           break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC32X64[DSTOFF] = (float32_t)
           (((int64_t)ALURegA.data.FLOAT32) OP
           ((int64_t)ALURegB.data.FLOAT32));
           break;\
} break;\
case 0xC0: switch(dtype)\
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC64X32[DSTOFF] = (ALURegA.data
           .UINT64) OP (ALURegB.data.UINT64);
           break;\
        case 0x20: DLARFile[DST].data.
           INTVEC64X32[DSTOFF] = (ALURegA.data.
           INT64) OP (ALURegB.data.INT64);
           break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC64X32[DSTOFF] = (float64_t)
           (((int64_t)ALURegA.data.FLOAT64) OP
           ((int64_t)ALURegB.data.FLOAT64));
           break;\
} break;\
```

```
                }\
                return 0;\
        }\
        else\
        {return 1;}\
}\



//Macro'd prototype for vector ALU operation
#define VALUOP(OPNAME, OP, DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
int do##OPNAME##v(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DESTOFF, uint8_t IMM)\
{\
        int i;\
        uint8_t dsize = pow(2, ((DLARFile[dst].flags & WDSZMASK)+3));\
        uint8_t dtype = DLARFile[dst].flags&TYPMASK;\
        DLARFile[DST].flags &= 0x08;\
        if ((typeconvertv(&ALULARA, DST , SRC1, OFF1) == 0) && (
           typeconvertv(&ALULARB, DST , SRC2, OFF2) == 0))\
        {\
                for(i=0;i<LARWIDTH/dsize;i++)\
                {\
                        DLAROPTYPV(DST, i, OP)\
                }\
                return 0;\
        }\
        else\
        {\
                return 1;\
        }\
}\


//Macro'd prototype for vector ALU operations that C doesn't like doing
   on non-integer bit patterns
#define VALUBINOP(OPNAME, OP, DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)
   \
int do##OPNAME##v(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DSTOFF, uint8_t IMM)\
{\
        int i;\
        uint8_t dsize = pow(2, ((DLARFile[DST].flags & 0xC0)+3));\
        uint8_t dtype = DLARFile[DST].flags&0x30; DLARFile[DST].flags &=
             0x08;\
        if ((typeconvertv(&ALULARA, DST , SRC1, OFF1) == 0) && (
           typeconvertv(&ALULARB, DST , SRC2, OFF2) == 0))\
        {\
                for(i=0;i<2048/dsize;i++)\
                {\
                        switch(dsize)\
                        {\
                                case 0x00: switch(dtype)\
```

```
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC8X256[i] = ALULARA.
           data.UINTVEC8X256[i] OP
           ALULARB.data.UINTVEC8X256[i
           ];break;\
        case 0x20: DLARFile[DST].data.
           INTVEC8X256[i] = ALULARA.
           data.INTVEC8X256[i] OP
           ALULARB.data.INTVEC8X256[i];
           break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC8X256[i] = (
           float64_t)((int64_t)ALULARA.
           data.FLOATVEC8X256[i] OP (
           int64_t)ALULARB.data.
           FLOATVEC8X256[i]);break;\
} break;\
case 0x40: switch(dtype)\
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC16X128[i] = ALULARA.
           data.UINTVEC16X128[i] OP
           ALULARB.data.UINTVEC16X128[i
           ];break;\
        case 0x20: DLARFile[DST].data.
           INTVEC16X128[i] = ALULARA.
           data.INTVEC16X128[i] OP
           ALULARB.data.INTVEC16X128[i
           ];break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC16X128[i] = (
           float64_t)((int64_t)ALULARA.
           data.FLOATVEC16X128[i] OP (
           int64_t)ALULARB.data.
           FLOATVEC16X128[i]);break;\
} break;\
case 0x80: switch(dtype)\
{\
        case 0x10: DLARFile[DST].data.
           UINTVEC32X64[i] = ALULARA.
           data.UINTVEC32X64[i] OP
           ALULARB.data.UINTVEC32X64[i
           ];break;\
        case 0x20: DLARFile[DST].data.
           INTVEC32X64[i] = ALULARA.
           data.INTVEC32X64[i] OP
           ALULARB.data.INTVEC32X64[i];
           break;\
        case 0x30: DLARFile[DST].data.
           FLOATVEC32X64[i] = (
           float64_t)((int64_t)ALULARA.
           data.FLOATVEC32X64[i] OP (
           int64_t)ALULARB.data.
```

```
                                                FLOATVEC32X64[i]);break;\
                                        } break; case 0xC0: switch(dtype)\
                                        {\
                                                case 0x10: DLARFile[DST].data.
                                                   UINTVEC64X32[i] = ALULARA.
                                                   data.UINTVEC64X32[i] OP
                                                   ALULARB.data.UINTVEC64X32[i
                                                   ];break;\
                                                case 0x20: DLARFile[DST].data.
                                                   INTVEC64X32[i] = ALULARA.
                                                   data.INTVEC64X32[i] OP
                                                   ALULARB.data.INTVEC64X32[i];
                                                   break;\
                                                case 0x30: DLARFile[DST].data.
                                                   FLOATVEC64X32[i] = (
                                                   float64_t)((int64_t)ALULARA.
                                                   data.FLOATVEC64X32[i] OP (
                                                   int64_t)ALULARB.data.
                                                   FLOATVEC64X32[i]);break;\
                                        } break;\
                                }\
                        }\
                        return 0;\
                }\
                else \
                {return 1;}\
}\



//A Sub-Macro for resolving typing
#define DLAROPTYPV(DST, OFF, OP)\
switch(dsize)\
{\
        case SZ8:\
                switch(dtype)\
                {\
                case TUINT:\
                        DLAROPDOV(UINTVEC8X256,DST, OFF, OP)\
                        break;\
                case TINT:\
                        DLAROPDOV(INTVEC8X256,DST, OFF, OP)\
                        break;\
                case TFLOAT:\
                        DLAROPDOV(FLOATVEC8X256,DST, OFF, OP) \
                        break;\
                }\
        break;\
        case SZ16:\
                switch(dtype)\
                {\
                        case TUINT:\
                                DLAROPDOV(UINTVEC16X128,DST, OFF, OP)\
```

```
                                break;\
                        case TINT:\
                                DLAROPDOV(INTVEC16X128,DST, OFF, OP)\
                                break;\
                        case TFLOAT:\
                                DLAROPDOV(FLOATVEC16X128,DST, OFF, OP) \
                                break;\
                }\
        break;\
        case SZ32: \
                switch(dtype)\
                {\
                        case TUINT:\
                                DLAROPDOV(UINTVEC32X64,DST, OFF, OP)
                                    break;\
                        case TINT:\
                                DLAROPDOV(INTVEC32X64, DST, OFF, OP)
                                    break;\
                        case TFLOAT:\
                                DLAROPDOV(FLOATVEC32X64, DST, OFF, OP)
                                    break;\
                }\
        break;\
        case SZ64: \
                switch(dtype)\
                {\
                        case TUINT:\
                                DLAROPDOV(UINTVEC64X32, DST, OFF, OP)
                                    break;\
                        case TINT:\
                                DLAROPDOV(INTVEC64X32, DST, OFF, OP)
                                    break;\
                        case TFLOAT:\
                                DLAROPDOV(FLOATVEC64X32, DST, OFF, OP)
                                    break;\
                }\
        break;\
}\


//A Sub-Macro for setting up operation typing; scalar operation version
#define DLAROPTYPS(OFF1, OFF2, DST, DSTOFF, OP)\
switch(dsize)\
{\
        case SZ8:\
                switch(dtype)\
                {\
                case TUINT:\
                        DLAROPDOS(UINT8, UINTVEC8X256, OP, OFF1, OFF2,
                            DST, DSTOFF)\
                        break;\
                case TINT:\
                        DLAROPDOS(INT8, INTVEC8X256, OP, OFF1, OFF2, DST
                            , DSTOFF)\
```

```
                break;\
            case TFLOAT:\
                DLAROPDOS(FLOAT8, FLOATVEC8X256, OP, OFF1, OFF2,
                     DST, DSTOFF) \
                break;\
            }\
        break;\
        case SZ16:\
            switch(dtype)\
            {\
                case TUINT:\
                    DLAROPDOS(UINT16, UINTVEC16X128, OP,
                        OFF1, OFF2, DST, DSTOFF)\
                    break;\
                case TINT:\
                    DLAROPDOS(INT16, INTVEC16X128, OP, OFF1,
                         OFF2, DST, DSTOFF)\
                    break;\
                case TFLOAT:\
                    DLAROPDOS(FLOAT16, FLOATVEC16X128, OP,
                        OFF1, OFF2, DST, DSTOFF)\
                    break;\
            }\
        break;\
        case SZ32: \
            switch(dtype)\
            {\
                case TUINT:\
                    DLAROPDOS(UINT32, UINTVEC64X32, OP, OFF1
                        , OFF2, DST, DSTOFF) break;\
                case TINT:\
                    DLAROPDOS(INT32, INTVEC32X64, OP, OFF1,
                        OFF2, DST, DSTOFF) break;\
                case TFLOAT:\
                    DLAROPDOS(FLOAT32, FLOATVEC32X64, OP,
                        OFF1, OFF2, DST, DSTOFF) break;\
            }\
        break;\
        case SZ64: \
            switch(dtype)\
            {\
                case TUINT:\
                    DLAROPDOS(UINT64, UINTVEC64X32, OP, OFF1
                        , OFF2, DST, DSTOFF) break;\
                case TINT:\
                    DLAROPDOS(INT64, INTVEC64X32, OP, OFF1,
                        OFF2, DST, DSTOFF) break;\
                case TFLOAT:\
                    DLAROPDOS(FLOAT64, FLOATVEC64X32, OP,
                        OFF1, OFF2, DST, DSTOFF) break;\
            }\
        break;\
}\
```

```
//This actually performs the operation once the types are set up - just
    saves some typing...
//MAY not need to pass OFF in, but better safe than eaten by the
    preprocessor.
#define DLAROPDOV(TYP, DST, OFF, OP)\
DLARFile[DST].data.TYP[OFF] = ALULARA.data.TYP[OFF] OP ALULARB.data.TYP[
    OFF];\

#define DLAROPDOS(STYP, VTYP, OP, OFF1, OFF2, DST, DSTOFF)\
DLARFile[DST].data.VTYP[DSTOFF] = (ALURegA.data.STYP) OP (ALURegB.data.
    STYP);\
```

Listing A.6: memops.h

```
// memops.h
// Macros for Data Memory Operations

//These are inherently vector operations...
//TYP is the mnemonic type (a number+letter)
//ETYP is the encoded type (a number)

//Store copies data between LARs, while manipulates the tags
#define STOP(TYP, ETYP, DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
int doSTORE##TYP(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DESTOFF, uint8_t IMM)\
{\
int address;\
ADDRCALC(DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
DLARFile[DST].addr=address;\
DLARFile[DST].flags=ETYP | ((DLARFile[SRC1].flags & DIRTYMASK));\
typeconvertv(&DLARFile[DST], DST , SRC1, OFF1);\
}\


#define LDOP(TYP, ETYP, DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
int doLOAD##TYP(uint8_t DST, uint8_t SRC1, uint8_t SRC2, uint8_t OFF1,
    uint8_t OFF2, uint8_t DESTOFF, uint8_t IMM)\
{\
int address;\
ADDRCALC(DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
DLARFile[DST].addr=address;\
DLARFile[DST].flags=ETYP;\
int i;\
if(ETYP==0x10)\
        for(i=0;i<256;i++)\
        {\
                DLARFile[DST].data.UINTVEC8X256[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x50)\
        for(i=0;i<128;i++)\
        {\
                DLARFile[DST].data.UINTVEC16X128[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x90)\
```

```
        for(i=0;i<64;i++)\
        {\
                DLARFile[DST].data.UINTVEC32X64[i]=RAMFile[address+i];\
        }\
else if(ETYP==0xD0)\
        for(i=0;i<32;i++)\
        {\
                DLARFile[DST].data.UINTVEC64X32[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x20)\
        for(i=0;i<256;i++)\
        {\
                DLARFile[DST].data.INTVEC8X256[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x60)\
        for(i=0;i<128;i++)\
        {\
                DLARFile[DST].data.INTVEC16X128[i]=RAMFile[address+i];\
        }\
else if(ETYP==0xC0)\
        for(i=0;i<64;i++)\
        {\
                DLARFile[DST].data.INTVEC32X64[i]=RAMFile[address+i];\
        }\
else if(ETYP==0xE0)\
        for(i=0;i<32;i++)\
        {\
                DLARFile[DST].data.INTVEC64X32[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x30)\
        for(i=0;i<256;i++)\
        {\
                DLARFile[DST].data.FLOATVEC8X256[i]=RAMFile[address+i];\
        }\
else if(ETYP==0x70)\
        for(i=0;i<128;i++)\
        {\
                DLARFile[DST].data.FLOATVEC16X128[i]=RAMFile[address+i
                   ];\
        }\
else if(ETYP==0xB0)\
        for(i=0;i<64;i++)\
        {\
                DLARFile[DST].data.FLOATVEC32X64[i]=RAMFile[address+i];\
        }\
else if(ETYP==0xF0)\
        for(i=0;i<32;i++)\
        {\
                DLARFile[DST].data.FLOATVEC64X32[i]=RAMFile[address+i];\
        }\
}\


//A special macro for calculating addresses with LARK's weird-ass
```

```
   addressing mode
// This should probably do a bounds check against RAMSZ
#define ADDRCALC(DST, SRC1, SRC2, OFF1, OFF2, DESTOFF, IMM)\
int srctyp=DLARFile[SRC2].flags&TYPMASK;\
switch(DLARFile[(SRC2)].flags & WDSZMASK)\
{\
        case SZ8:\
                switch(srctyp)\
                {\
                        case TUINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.UINTVEC8X256[(OFF2)]+
                                   (IMM*pow(2, ((DLARFile[DST].flags &
                                   WDSZMASK)+3)))); break;\
                        case TINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.INTVEC8X256[(OFF2)]+ (
                                   IMM*pow(2, ((DLARFile[DST].flags &
                                   WDSZMASK)+3)))); break;\
                        case TFLOAT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.FLOATVEC8X256[(OFF2)]+
                                    (IMM*pow(2, ((DLARFile[DST].flags &
                                    WDSZMASK)+3)))); break;\
                }\
        break;\
        case SZ16:\
                switch(srctyp)\
                {\
                        case TUINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.UINTVEC16X128[(OFF2)]+
                                    (IMM*pow(2, ((DLARFile[DST].flags &
                                    WDSZMASK)+3)))); break;\
                        case TINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.INTVEC16X128[(OFF2)]+
                                   (IMM*pow(2, ((DLARFile[DST].flags &
                                   WDSZMASK)+3)))); break;\
                        case TFLOAT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.FLOATVEC16X128[(OFF2)
                                   ]+ (IMM*pow(2, ((DLARFile[DST].flags
                                    & WDSZMASK)+3)))); break;\
                }\
        break;\
        case SZ32:\
                switch(srctyp)\
                {\
                        case TUINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                   [(SRC2)].data.UINTVEC32X64[(OFF2)]+
                                   (IMM*pow(2, ((DLARFile[DST].flags &
                                   WDSZMASK)+3)))); break;\
```

```
                        case TINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                    [(SRC2)].data.INTVEC32X64[(OFF2)]+ (
                                    IMM*pow(2, ((DLARFile[DST].flags &
                                    WDSZMASK)+3)))); break;\
                        case TFLOAT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                    [(SRC2)].data.FLOATVEC32X64[(OFF2)]+
                                     (IMM*pow(2, ((DLARFile[DST].flags &
                                     WDSZMASK)+3)))); break;\
                }\
        break;\
        case SZ64: \
                switch(srctyp)\
                {\
                        case TUINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                    [(SRC2)].data.UINTVEC64X32[(OFF2)]+
                                    (IMM*pow(2, ((DLARFile[DST].flags &
                                    WDSZMASK)+3)))); break;\
                        case TINT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                    [(SRC2)].data.INTVEC64X32[(OFF2)]+ (
                                    IMM*pow(2, ((DLARFile[DST].flags &
                                    WDSZMASK)+3)))); break;\
                        case TFLOAT:\
                                address= (DLARFile[SRC1].addr + DLARFile
                                    [(SRC2)].data.FLOATVEC64X32[(OFF2)]+
                                     (IMM*pow(2, ((DLARFile[DST].flags &
                                     WDSZMASK)+3)))); break;\
                }\
        break;\
}\
```

Appendix B lark1.aik

Listing B.1: lark1.aik

```
;I had a confusion with the aliasing, so just replicating with regexes
    ...

;Yah, there are a lot of DLARS in this design...
.const 0 L0 L1 L2 L3 L4 L5 L6 L7 L8 L9 L10 L11 L12 L13 L14 L15 L16 L17
    L18 L19 L20 L21 L22 L23 L24 L25 L26 L27 L28 L29 L30 L31 L32 L33 L34
    L35 L36 L37 L38 L39 L40 L41 L42 L43 L44 L45 L46 L47 L48 L49 L50 L51
    L52 L53 L54 L55 L56 L57 L58 L59 L60 L61 L62 L63 L64 L65 L66 L67 L68
    L69 L70 L71 L72 L73 L74 L75 L76 L77 L78 L79 L80 L81 L82 L83 L84 L85
    L86 L87 L88 L89 L90 L91 L92 L93 L94 L95 L96 L97 L98 L99 L100 L101
    L102 L103 L104 L105 L106 L107 L108 L109 L110 L111 L112 L113 L114
    L115 L116 L117 L118 L119 L120 L121 L122 L123 L124 L125 L126 L127
    L128 L129 L130 L131 L132 L133 L134 L135 L136 L137 L138 L139 L140
    L141 L142 L143 L144 L145 L146 L147 L148 L149 L150 L151 L152 L153
    L154 L155 L156 L157 L158 L159 L160 L161 L162 L163 L164 L165 L166
    L167 L168 L169 L170 L171 L172 L173 L174 L175 L176 L177 L178 L179
    L180 L181 L182 L183 L184 L185 L186 L187 L188 L189 L190 L191 L192
    L193 L194 L195 L196 L197 L198 L199 L200 L201 L202 L203 L204 L205
    L206 L207 L208 L209 L210 L211 L212 L213 L214 L215 L216 L217 L218
    L219 L220 L221 L222 L223 L224 L225 L226 L227 L228 L229 L230 L231
    L232 L233 L234 L235 L236 L237 L238 L239 L240 L241 L242 L243 L244
    L245 L246 L247 L248 L249 L250 L251 L252 L253 L254 L255;

;LOADs
LOAD8U dst src1 src2 imm := 0x44:8 dst:8 src1:8 src2:8 imm:32;
LOAD16U dst src1 src2 imm := 0x45:8 dst:8 src1:8 src2:8 imm:32;
LOAD32U dst src1 src2 imm := 0x46:8 dst:8 src1:8 src2:8 imm:32;
LOAD64U dst src1 src2 imm := 0x47:8 dst:8 src1:8 src2:8 imm:32;
LOAD8I dst src1 src2 imm := 0x48:8 dst:8 src1:8 src2:8 imm:32;
LOAD16I dst src1 src2 imm := 0x49:8 dst:8 src1:8 src2:8 imm:32;
LOAD32I dst src1 src2 imm := 0x4A:8 dst:8 src1:8 src2:8 imm:32;
LOAD64I dst src1 src2 imm := 0x4B:8 dst:8 src1:8 src2:8 imm:32;
LOAD8F dst src1 src2 imm := 0x4C:8 dst:8 src1:8 src2:8 imm:32;
LOAD16F dst src1 src2 imm := 0x4D:8 dst:8 src1:8 src2:8 imm:32;
LOAD32F dst src1 src2 imm := 0x4E:8 dst:8 src1:8 src2:8 imm:32;
LOAD64F dst src1 src2 imm := 0x4F:8 dst:8 src1:8 src2:8 imm:32;

;Stores
STORE8U dst src1 src2 imm := 0x54:8 dst:8 src1:8 src2:8 imm:32;
STORE16U dst src1 src2 imm := 0x55:8 dst:8 src1:8 src2:8 imm:32;
STORE32U dst src1 src2 imm := 0x56:8 dst:8 src1:8 src2:8 imm:32;
STORE64U dst src1 src2 imm := 0x57:8 dst:8 src1:8 src2:8 imm:32;
STORE8I dst src1 src2 imm := 0x58:8 dst:8 src1:8 src2:8 imm:32;
STORE16I dst src1 src2 imm := 0x59:8 dst:8 src1:8 src2:8 imm:32;
STORE32I dst src1 src2 imm := 0x5A:8 dst:8 src1:8 src2:8 imm:32;
STORE64I dst src1 src2 imm := 0x5B:8 dst:8 src1:8 src2:8 imm:32;
```

```
STORE8F dst src1 src2 imm := 0x5C:8 dst:8 src1:8 src2:8 imm:32;
STORE16F dst src1 src2 imm := 0x5D:8 dst:8 src1:8 src2:8 imm:32;
STORE32F dst src1 src2 imm := 0x5E:8 dst:8 src1:8 src2:8 imm:32;
STORE64F dst src1 src2 imm := 0x5F:8 dst:8 src1:8 src2:8 imm:32;



;ALU Operations
ADDs dst src1 src2 off1 off2 destoff imm := 0x80:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SUBs dst src1 src2 off1 off2 destoff imm := 0x81:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
MULs dst src1 src2 off1 off2 destoff imm := 0x82:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
DIVs dst src1 src2 off1 off2 destoff imm := 0x83:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
MODs dst src1 src2 off1 off2 destoff imm := 0x84:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
ANDs dst src1 src2 off1 off2 destoff imm := 0x85:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
ORs dst src1 src2 off1 off2 destoff imm := 0x86:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
XORs dst src1 src2 off1 off2 destoff imm := 0x87:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
NOTs dst src1 src2 off1 off2 destoff imm := 0x88:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SLLs dst src1 src2 off1 off2 destoff imm := 0x89:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SRAs dst src1 src2 off1 off2 destoff imm := 0x8A:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SRLs dst src1 src2 off1 off2 destoff imm := 0x8B:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SLTs dst src1 src2 off1 off2 destoff imm := 0x8C:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;

ADDv dst src1 src2 off1 off2 destoff imm := 0xA0:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SUBv dst src1 src2 off1 off2 destoff imm := 0xA1:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
MULv dst src1 src2 off1 off2 destoff imm := 0xA2:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
DIVv dst src1 src2 off1 off2 destoff imm := 0xA3:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
MODv dst src1 src2 off1 off2 destoff imm := 0xA4:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
ANDv dst src1 src2 off1 off2 destoff imm := 0xA5:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
ORv dst src1 src2 off1 off2 destoff imm := 0xA6:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
XORv dst src1 src2 off1 off2 destoff imm := 0xA7:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
NOTv dst src1 src2 off1 off2 destoff imm := 0xA8:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SLLv dst src1 src2 off1 off2 destoff imm := 0xA9:8 dst:8 src1:8 src2:8
```

```
    off1:8 off2:8 destoff:8;
SRAv dst src1 src2 off1 off2 destoff imm := 0xAA:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SRLv dst src1 src2 off1 off2 destoff imm := 0xAB:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;
SLTv dst src1 src2 off1 off2 destoff imm := 0xAC:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8;


;Flow control

;SEL is actually the same as ARITH, but interpreted a little differently
    .
SEL op dst src1 src2 off1 off2 destoff imm:=0xC0:8 dst:8 src1:8 src2:8
    off1:8 off2:8 destoff:8 imm:8;
; How the HELL do call and return get shoveled into the not-ILAR stuff
    ...
CALL;
RETURN;

;using 00 as an opcode is a little sketch, but it is the one required
    for boostraping
FETCH dest src1 src2 num imm := 0x00:8 dest:8 src1:8 src2:8 num:4 imm
    :28;
```

Appendix C larc

Listing C.1: larcast.g

```
/*      LARC Compiler
 *      Paul S. Eberhart, 2011-04-15
 *      Borrows somewhat from the MIMDC interpreter by Hank Dietz and
    Will Cohen
 *      and the (C++ mode) public domain (Terrence Parr/Randy McRee/Tory
     Eneboe) ANSI C grammar provided as an example.
 *      Implements a C-like language to target a LAR-based architecture
*/

#header <<
#include <sys/types.h>
#include <math.h>
#include <string.h>
#define TOKBUF 24

//A datastructure for the attributes

typedef struct
{
        // 1 for int, 2 for float, 3 for literal... 0 is a debug case
        int typ;
        union {float f; int i; char t[TOKBUF];} value;
} Attrib;

#define AST_FIELDS Attrib data;

//#define zzcr_ast(ast,attr,tok,text) memcpy(&(ast->data), attr, sizeof
   (*attr));

#define zzcr_ast(ast,attr,tok,text) ast->data.value = *attr.value; ast->
   data.typ=*attr.typ;

>>




//LARC Predefined tokens

//Types
#token UINT_8T "t_8u"
#token UINT_16T "t_16u"
#token UINT_32T "t_32u"
#token UINT_64T "t_64u"
#token INT_8T "t_8i"
#token INT_16T "t_16i"
#token INT_32T "t_32i"
```

```
#token INT_64T "t_64i"
#token FLOAT_8T "t_8f"
#token FLOAT_16T "t_16f"
#token FLOAT_32T "t_32f"
#token FLOAT_64T "t_64f"
#token UINTVEC8X256 "v_8u256"
#token UINTVEC16X128 "v_16u128"
#token UINTVEC32X64 "v_32u64"
#token UINTVEC64X32 "v_64u32"
#token INTVEC8X256 "v_8i256"
#token INTVEC16X128 "v_16i128"
#token INTVEC32X64 "v_32i64"
#token INTVEC64X32 "v_64i32"
#token FLOATVEC8X256 "v_8f256"
#token FLOATVEC16X128 "v_16f128"
#token FLOATVEC32X64 "v_32f64"
#token FLOATVEC64X32 "v_64f32"

//Operators
#token ASSIGN "="
#token EQUAL "=="
#token LESSTHAN "\<"
#token PLUS "\+"
#token MINUS "\-"
#token STAR "\*"
#token DIVIDE "/"
#token MOD "\%"
#token BITWISEOR "\|"
#token BITWISEXOR "^"
#token AMPERSAND "&"
#token NOT "\~"
#token RTSHIFT "\<\<"
#token LFTSHIFT "\>\>"


//Keywords
#token RETURN "return"
#token WHILE "while"
#token IF "if"
#token ELSE "else"

//Misc. Tokens and classes of literal
#token LCURLYBRACE "\{"
#token RCURLYBRACE "\}"
#token LPAREN "\("
#token RPAREN "\)"
#token SEMICOLON ";"
#token OCTALINT "0[0-7]*{[uUlL]}"
#token DECIMALINT "[1-9][0-9]*{[uUlL]}"
#token HEXINT "(0x|0X)[0-9a-fA-F]+{[uUlL]}"
#token FNUM1   "[0-9]+.[0-9]*{[Ee]{\+|\-}[0-9]+}"
                        //<< val.data.f = atof(zzlextext); >>
#token FNUM2   "[1-9][0-9]*[Ee]{\+|\-}[0-9]+"
                        //<< val.f = atof(zzlextext); >>
```

```
#token FNUM3   ".[0-9]+{[Ee]{\+|\-}[0-9]+}"
                        //<< val.data.f = atof(zzlextext); >>
#token "'" << zzmode(CHARACTERS); zzmore(); >>
#token "\"" << zzmode(STRINGS); zzmore(); >>
#token IDENTIFIER "[a-zA-Z_][a-zA-Z0-9_]*"


//Whitespace and such
#token Eof "@"
#token SPACE "[\ ]+"    <<zzskip();>>
#token TAB "[\t]+"      <<zzskip();>>
#token NEWLINE "[\n]"   << zzskip(); zzline++; >>




//Comments
#token "\/\*" << zzmode(COMMENTS); zzskip(); >>
#token "\/\/" <<zzmode(SLCOMMENT); zzskip();>>
#lexclass SLCOMMENT
#token "~[\n]" <<zzskip();>>
#token "[\n]" <<zzmode(START); zzskip(); zzline++;>>




#lexclass COMMENTS
#token "[\n\r]"         << zzskip(); zzline++; >>
#token "~[\n]" <<zzskip();>>
#token "\*\/"           << zzmode(START); zzskip (); >>
#token "\/\*"           << zzskip(); >>
#token "~[\*\n\r]+"     << zzskip(); >>



#lexclass STRINGS

#token STRING "\"" << zzmode(STRINGS); >>
#token "\\n" << zzreplchar ((char) 0x0A); zzmore(); >>
#token "\\t" << zzreplchar ((char) 0x09); zzmore(); >>
#token "\\v" << zzreplchar ((char) 0x0B); zzmore(); >>
#token "\\b" << zzreplchar ((char) 0x08); zzmore(); >>
#token "\\r" << zzreplchar ((char) 0x0D); zzmore(); >>
#token "\\f" << zzreplchar ((char) 0x0C); zzmore(); >>
#token "\\a" << zzreplchar ((char) 0x07); zzmore(); >>
#token "\\\\" << zzreplchar ((char) 0x5C); zzmore(); >>
#token "\\?" << zzreplchar ((char) 0x3F); zzmore(); >>
#token "\\'" << zzreplchar ((char) 0x27); zzmore(); >>
#token "\\\"" << zzreplchar ((char) 0x22); zzmore(); >>
#token "\\0[0-7]*" << zzreplchar ((char) strtol (zzbegexpr+1, NULL, 8));
     zzmore(); >>
#token "\\[1-9][0-9]*" << zzreplchar ((char) strtol (zzbegexpr+1, NULL,
    10)); zzmore(); >>
#token "\\(0x|0X)[0-9a-fA-F]+" << zzreplchar ((char) strtol (zzbegexpr
```

```
    +1, NULL, 16)); zzmore(); >>
#token "[\n\r]" << zzline++; zzmore(); >>
#token "~[\"\n\r\\]+" << zzmore(); >>

#lexclass CHARACTERS

//Deal with escaped characters.
#token CHARACTER "'" << zzmode(START); >>
#token "\\n" << zzreplchar((char) 0x0A); zzmore(); zzmode(DONE); >>
#token "\\t" << zzreplchar((char) 0x09); zzmore(); zzmode(DONE); >>
#token "\\v" << zzreplchar((char) 0x0B); zzmore(); zzmode(DONE); >>
#token "\\b" << zzreplchar((char) 0x08); zzmore(); zzmode(DONE); >>
#token "\\r" << zzreplchar((char) 0x0D); zzmore(); zzmode(DONE); >>
#token "\\f" << zzreplchar((char) 0x0C); zzmore(); zzmode(DONE); >>
#token "\\a" << zzreplchar((char) 0x07); zzmore(); zzmode(DONE); >>
#token "\\\\" << zzreplchar((char) 0x5C); zzmore(); zzmode(DONE); >>
#token "\\?" << zzreplchar((char) 0x3F); zzmore(); zzmode(DONE); >>
#token "\\'" << zzreplchar((char) 0x27); zzmore(); zzmode(DONE); >>
#token "\\\"" << zzreplchar((char) 0x22); zzmore(); zzmode(DONE); >>
#token "\\0[0-7]*" << zzreplchar((char) strtol (zzbegexpr+1, NULL, 8));
    zzmore(); zzmode(DONE); >>
#token "\\[1-9][0-9]*" << zzreplchar((char) strtol (zzbegexpr+1, NULL,
    10)); zzmore(); zzmode(DONE); >>
#token "\\(0x|0X)[0-9a-fA-F]+" << zzreplchar((char) strtol (zzbegexpr+1,
     NULL, 16)); zzmore(); zzmode(DONE); >>
#token "[\n\r]" << zzline++; zzmore(); >>
#token "~['\n\r\\]" << zzmore(); zzmode(DONE); >>

#lexclass DONE


#lexclass START


//C Preamble
<<
/* Stuff for the printing of ASTs */
void show(tree)
AST *tree;
{
        printf("[");
        switch(tree->data.typ)
        {
        case 1: printf("%d", tree->data.value.i); break;
        case 2: printf("%f", tree->data.value.f); break;
        case 3: printf("%s", tree->data.value.t); break;
        default: printf("%s", tree->data.value.t); break;
        }
        printf(",%d] ",tree->data.typ);
}
void before() { printf(" ("); }
void after() { printf(") "); }
```

```
zzcr_attr(Attrib *attr, int token, char *text)
{
        switch(token)
        {
                //Floats
                case FNUM1:
                case FNUM2:
                case FNUM3: attr->value.f = atof(text); attr->typ = 2;
                   break;
                //Ints
                case OCTALINT:
                case HEXINT:
                case DECIMALINT: attr->value.i = atoi(text); attr->typ =
                    1; break;
                //String data
                case STRING: strcpy(attr->value.t,text); attr->typ = 3;
                   break;
                default: strcpy(attr->value.t, text); attr->typ = -1;
        }
}

zzd_attr(Attrib* attr)
{
        if ((attr->typ == 3) && (attr!=NULL))
        {
                free(attr); attr=NULL;
        }
}

/*
larcast(AST *t, Attrib *a, int tok, char *text)
{
        //Make a copy of the Attrib for the tree
        memcpy(t->data ,a, sizeof(*a));
}
*/


AST *root = NULL;
//A Main
main()
{
        printf("Calling ANTLR\n");
        ANTLR(prog(&root), stdin);
        printf("Input accepted! \n");
        printf("Printing tree, format is [Token,Type], parens indicate
           nesting.\n");
        zzpre_ast(root,show,before,after);
        printf("\n");
}

>>

prog: (decl)* (func)* "@" <<printf("Recognized program \n");>>
```

```
;

//The LARC type system is kind of complicated, becuase it supports every
     native type on the LARK1 spec, including native-width vectors.
typ: UINT_8T | UINT_16T | UINT_32T | UINT_64T |
        INT_8T | INT_16T | INT_32T | INT_64T |
        FLOAT_8T | FLOAT_16T | FLOAT_32T | FLOAT_64T |
        UINTVEC8X256 | UINTVEC16X128 | UINTVEC32X64 | UINTVEC64X32 |
        INTVEC8X256 | INTVEC16X128 | INTVEC32X64 | INTVEC64X32 |
        FLOATVEC8X256 | FLOATVEC16X128 | FLOATVEC32X64 | FLOATVEC64X32
;

//Declarations of functions vs. variables are factored strangely to
   avoid ambiguity
decl:
        typ (IDENTIFIER) (vardec | func)
;

//Later, specifers will be needed to be able to do I/O and suchlike.
   For now type id (name)*
vardec:
        (","IDENTIFIER)* ";"
        <<printf("Found a declaration\n");>>
;

//Right now a totally empty function is OK
func: "\(" {typ IDENTIFIER ( "," typ IDENTIFIER)*} "\)" stat
<<printf("Found a function definition\n");>>
;

stat:
        "\{"(decl)* (stat)* "\}" |
        RETURN { expr } ";"|
        WHILE "\(" expr "\)" stat |
        IF "\(" expr "\)" stat { ELSE stat } |
        expr ";" |
        ";"
;


//Handles Expressions, Broken down by type.  Order = precedence, trying
   to match K&R C where convienent.
// PSE20110505: This was in here for some reason I can't fathom... (","
   assign_e)
//              PSE20120131 -- Derp. That is the comma operator.
   Replaced.
expr: assign_e ("," assign_e)*
;

//Initalizing native vector with "v_8u256 a = [0,1,2,3,4];" should do
   some counting
//  - make sure it fits and pad extras with 0?
assign_e:  or_e {"="^ expr}
;
```

```
or_e: xor_e ("\|"^ xor_e)*
;

xor_e: and_e ("^"^ and_e)*
;

and_e: rel_e ("&"^ rel_e)*
;

rel_e: shft_e (("<"^ | "=="^) shft_e)*
;

//How is the arithmetic v. logical right context handled?
shft_e: addsub_e (("\<\<"^ | "\>\>"^) addsub_e)*
;


addsub_e: muldiv_e (("\+"^ | "\-"^) muldiv_e)*
;

muldiv_e: neg_e (("\*"^ | "/"^ | "\%"^) neg_e)*
;


neg_e: "\~"^ neg_e
     | cast_e
;

//This is going to be ugly behind the scenes -- vector-to-vector casts!
cast_e: {"\(" typ "\)"} primary_e
;

primary_e: IDENTIFIER | num | vec
;

//The many sorts of constant are down here.
num: (int_c | float_c)
;

int_c: OCTALINT | DECIMALINT | HEXINT
;

float_c: FNUM1 | FNUM2 | FNUM3
;

//Doing the two separtately has an obvious prefix problem with "["
// Requires user to put at least one value in there to tag type.
vec: "\[" ((int_c)+ | (float_c)+) "\]";
```

Bibliography

[1] J. Backus, "Can programming be liberated from the von neumann style?: A functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, ISSN: 0001-0782. DOI: `10.1145/359576.359579`. [Online]. Available: `http://doi.acm.org/10.1145/359576.359579`.

[2] *Intel advanced vector extensions programming reference*, Intel Corporation, Apr. 2011. [Online]. Available: `https://software.intel.com/sites/default/files/4f/5b/36945`.

[3] *Intel itanium architecture software developer's manual*, Revision 2.3, Intel Corporation, May 2010.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems," in *Proceedings of the tenth international symposium on Hardware/software codesign*, ser. CODES '02, Estes Park, Colorado: ACM, 2002, pp. 73–78, ISBN: 1-58113-542-4. DOI: `10.1145/774789.774805`. [Online]. Available: `http://doi.acm.org/10.1145/774789.774805`.

[5] C. Carvalho, "The gap between processor and memory speeds," in *3rd Internal Conference on Computer Architecture (ICCA'02)*, A. J. Proenca, Ed., Braga, Portugal, Jan. 2002.

[6] H. Dietz and C. H. Chi, "Cregs: A new kind of memory for referencing arrays and pointers," in *Proceedings of Supercomputing'88*, Supercomputing, Jan. 1988, pp. 360–367.

[7] R. Fisher, "General- purpose simd within a register: Parallel processing on consumer microprocessors," PhD thesis, Purdue University, May 2003.

[8] G. A. Brent, "Using program structure to achieve prefetching for cache memories," UMI Order No. GAX87-21594, PhD thesis, University of Illinois, Department of Computer Science, Urbana, IL, USA, 1987.

[9] K. Melarkode, "Line associative registers," Master's thesis, University of Kentucky, Oct. 2004. [Online]. Available: `https://uknowledge.uky.edu/gradschool_theses/247/`.

[10] J. E. Smith, "Decoupled access/execute computer architectures," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, pp. 112–119, Apr. 1982, ISSN: 0163-5964. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1067649.801719`.

[11] R. J. Fisher and H. G. Dietz, "Compiling for simd within a register," in *Languages and Compilers for Parallel Computing: 11th International Workshop, LCPC'98 Chapel Hill, NC, USA, August 7–9, 1998 Proceedings*, S. Chatterjee, J. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P.-C. Yew, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 290–305, ISBN: 978-

3-540-48319-9. DOI: 10.1007/3-540-48319-5_19. [Online]. Available: http://dx.doi.org/10.1007/3-540-48319-5_19.

[12] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 248966-035. Nov. 2016.

[13] T. Diede, C. F. Hagenmaier, G. S. Miranker, J. J. Rubinstein, and W. S. Worley Jr., "The titan graphics supercomputer architecture," *Computer*, vol. 21, no. 9, pp. 13–28, 30, Sep. 1988, ISSN: 0018-9162. DOI: 10.1109/2.14344. [Online]. Available: http://dx.doi.org/10.1109/2.14344.

[14] K. Asanovic, *Risc-v vector extension proposal v0.2*, Berkeley, CA: 5th RISC-V Workshop, Nov. 2016.

[15] M. Weiss, "Strip mining on simd architectures," in *Proceedings of the 5th International Conference on Supercomputing*, ser. ICS '91, Cologne, West Germany: ACM, 1991, pp. 234–243, ISBN: 0-89791-434-1. DOI: 10.1145/109025.109083. [Online]. Available: http://doi.acm.org/10.1145/109025.109083.

[16] E. F. Gehringer and J. L. Keedy, "Tagged architecture: How compelling are its advantages?" *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 162–170, Jun. 1985, ISSN: 0163-5964. DOI: 10.1145/327070.327153. [Online]. Available: http://doi.acm.org/10.1145/327070.327153.

[17] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984, ISBN: 0932376223.

[18] A. J. W. Mayer, "The architecture of the burroughs b5000: 20 years later and still ahead of the times?" *SIGARCH Comput. Archit. News*, vol. 10, no. 4, pp. 3–10, Jun. 1982, ISSN: 0163-5964. DOI: 10.1145/641542.641543. [Online]. Available: http://doi.acm.org/10.1145/641542.641543.

[19] G. J. Myers and B. R. S. Buckingham, "A hardware implementation of capability-based addressing," *SIGARCH Comput. Archit. News*, vol. 8, no. 6, pp. 12–24, Oct. 1980, ISSN: 0163-5964. DOI: 10.1145/641914.641916. [Online]. Available: http://doi.acm.org/10.1145/641914.641916.

[20] D. A. Moon, "Architecture of the symbolics 3600," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 76–83, Jun. 1985, ISSN: 0163-5964. DOI: 10.1145/327070.327133. [Online]. Available: http://doi.acm.org/10.1145/327070.327133.

[21] *Intel iapx432 general data processor architecture reference manual*, Intel Corporation, Jan. 1981.

[22] R. P. Colwell, E. F. Gehringer, and E. D. Jensen, "Performance effects of architectural complexity in the intel 432," *ACM Trans. Comput. Syst.*, vol. 6, no. 3, pp. 296–339, Aug. 1988, ISSN: 0734-2071. DOI: 10.1145/45059.214411. [Online]. Available: http://doi.acm.org/10.1145/45059.214411.

[23] *Reference manual: Harris 800 general purpose digital computer*, Harris Corporation, Computer Systems Division, 2101 Cypress Creek Road, Fort Laudrdale, Flordia 33309, Aug. 1979.

[24] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan, “Compilation for explicitly managed memory hierarchies,” in *PPOPP'07*, 2007, pp. 226–236.

[25] *Intel 64 and ia-32 architectures software developer's manual - volume 2*, Intel Corporation, Sep. 2016. [Online]. Available: `https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf`.

[26] C. Harrison, *From the stacks: Programming the cache on the powerpc 750gx/fx*, IBM Corporation, Jan. 2005. [Online]. Available: `https://www.ibm.com/developerworks/library/pa-ppccache/pa-ppccache-pdf.pdf`.

[27] *Cortex-r4 and cortex-r4f technical reference manual*, ARM Limited, 2009. [Online]. Available: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0363e/Bgbfcaej.html`.

[28] M. Mock, R. Villamarin, and J. Baiocchi, “An empirical study of data speculation use on the intel itanium 2 processor,” in *9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*, Feb. 2005, pp. 22–33. DOI: `10.1109/INTERACT.2005.2`.

[29] N. Y. Lim, “Separating instruction fetches from memory accesses : Ilar (instruction line associative registers),” Master's thesis, University of Kentucky, May 2009. [Online]. Available: `https://uknowledge.uky.edu/gradschool_theses/625/`.

[30] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye, “Instruction fetch mechanisms for vliw architectures with compressed encodings,” in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, Dec. 1996, pp. 201–211. DOI: `10.1109/MICRO.1996.566462`.

[31] M. A. Sparks, “A comprehensive hdl model of a line associative register based architecture,” Master's thesis, University of Kentucky, 2013. [Online]. Available: `https://uknowledge.uky.edu/ece_etds/26/`.

[32] C. Lattner, “Llvm: An infrastructure for multi-stage optimization,” Master's thesis, University of Illinois at Urbana-Champaign, Dec. 2002.

[33] R. M. Stallman and the GCC Developer Community, “Using the gnu compiler collection version 4.6.0,” in. GNU Press, 2010, ch. 6.54, pp. 503–519.

[34] M. Woo and M. Brukman, *Writing an llvm compiler backend*, LLVM Project, 2012. [Online]. Available: `https://releases.llvm.org/3.1/docs/WritingAnLLVMBackend.html`.

[35] T. J. Parr, H. G. Dietz, and W. E. Cohen, “Pccts reference manual: Version 1.00,” *SIGPLAN Not.*, vol. 27, no. 2, pp. 88–165, Feb. 1992, ISSN: 0362-1340. DOI: `10.1145/130973.130980`. [Online]. Available: `http://doi.acm.org/10.1145/130973.130980`.

[36] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll(k) parser generator," *Softw. Pract. Exper.*, vol. 25, no. 7, pp. 789–810, Jul. 1995, ISSN: 0038-0644. DOI: 10.1002/spe.4380250705. [Online]. Available: http://dx.doi.org/10.1002/spe.4380250705.

[37] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *SIGPLAN Not.*, vol. 35, no. 5, pp. 145–156, May 2000, ISSN: 0362-1340. DOI: 10.1145/358438.349320. [Online]. Available: http://doi.acm.org/10.1145/358438.349320.

[38] P. Briggs, K. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," eng, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 428–455, 1994, ISSN: 1558-4593.

[39] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999, ISSN: 0164–0925. DOI: 10.1145/330249.330250. [Online]. Available: http://doi.acm.org/10.1145/330249.330250.

[40] H. Dietz and W. Dieter, "Aik, the assembler interpreter from kentucky," Tech. Rep., Apr. 2007. [Online]. Available: http://aggregate.org/AIK/aik20070415.pdf.

[41] N. Abu-Ghazaleh, D. Ponomarev, and D. Evtyushkin, "How the spectre and meltdown hacks really worked," *IEEE Spectrum*, vol. 56, no. 3, pp. 42–49, Mar. 2019, ISSN: 0018-9235. DOI: 10.1109/MSPEC.2019.8651934. [Online]. Available: https://spectrum.ieee.org/computing/hardware/how-the-spectre-and-meltdown-hacks-really-worked.

[42] T. R. Halfhill, "Transmeta breaks x86 low-power barrier," Tech. Rep., Feb. 2000.

[43] A. Fog, *The microarchitecture of intel, amd and via cpus*, 2018. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf.

Vita

Paul S. Eberhart was born and raised in Lexington, Kentucky. He earned Bachelor's degrees in Electrical Engineering, Computer Engineering, and Computer Science from the University of Kentucky in December 2008. In 2017 he won the Provost's Teaching Award from the University of Kentucky College of Engineering for his work in the Digital Logic Lab. Currently, Paul is pursuing a a PhD in Computer Science at the University of Kentucky.